An FPGA Coprocessor for Real-Time Bathymetric Synthetic Aperture Sonar

David J. Mulligan, B.E. (Hons)

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

February 2007

ABSTRACT

The following is a thesis for a Master's degree in Electrical Engineering. It presents the design of an FPGA coprocessor for real-time bathymetric synthetic aperture sonar. Bathymetry is the process of finding the height of the seafloor; a problem that requires the computation of a large number of short-length correlations and runs slowly on a conventional microprocessor architecture. It is desirable to generate the seafloor bathymetry in real time for use as a visual aid during data gathering, thus the development of a customised coprocessor is required.

The design presented utilises the system-on-chip (SoC) approach to FPGA programming, with a microprocessor, memory, communication cores and custom hardware all contained within a single chip. The merits of SoC design are examined and the details of this implementation are presented. The coprocessor communicates with a host computer over a USB link, receiving raw data as it is collected and sending processed data back to be displayed on-screen.

The system was successful as a proof-of-concept, capable of processing an eighth of the area imaged by the sonar in real-time. The results for a simulated scene are presented and the performance of the current system examined with a view to improving its capabilities. While further work is required to implement a complete solution to the problem, the work carried out thus far has provided a solid base for future research.

ACKNOWLEDGEMENTS

Firstly I would like to thank my supervisors, Dr Michael Hayes and Professor Peter Gough, for the guidance that has allowed me to complete this thesis. Michael's help in the more mathematical side of things especially proved invaluable. I would also like to thank Steve Weddell and Dr Andrew Bainbridge-Smith for all their help with understanding the finer points of VHDL and Xilinx FPGAs, as well as loaning expensive development boards for me to experiment on.

Secondly I would like to thank my peers in the Acoustics Research Group: Ed Pilbrow, Al Hunter and Phil Barclay for their help and discussions that aided the development of aspects of the design. Phil's extensive knowledge of InSAS was particularly useful.

Finally thank you to my parents for all their support and encouragement, and to my friends and flatmates for providing numerous distractions along the way.

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Currently very little is known about the terrain that lies beneath the ocean's surface, as obtaining high quality images of the sea floor is not an easy task. The high attenuation rate of light in sea water generally prohibits the use of optical methods, so other techniques must be used. Synthetic aperture sonar (SAS) is one such technique [1],[2]. A traditional downward-looking sonar has limited range and hence is slow to map an area of the seafloor. Side-looking sonar is thus employed to increase the range by projecting sound energy sideways through the medium. This requires the use of low frequencies to achieve long ranges, as the propagation of sound energy in water decreases with increasing frequency [3]. To achieve a desirable resolution with such low frequencies, an impractically large receiver aperture is required. A SAS system aims to solve this problem by moving a small aperture receiver along the scene and assembling the signals received into a large 'synthetic' aperture [1]. An advantage of this approach is that when coherent addition methods are applied to the received signals the resolution of the SAS image is independent of range [2, pp. 15–20].

Finding the bathymetry (height) of the seafloor requires an extension to the SAS method called interferometric SAS (InSAS). At least two vertically separated receiver arrays are employed and the received signals processed using standard SAS techniques. By comparing the SAS images, the time delay between the signals from each receiver can be calculated, allowing the height of the seafloor to be inferred at each pixel. Unfortunately SAS and InSAS require significant processing power; a modern high-end desktop PC can keep up with the demands of SAS processing, but performing InSAS techniques is not currently possible in real-time. The aim of this thesis is to develop a coprocessor capable of handling the extra data rate requirements of InSAS processing. In order to develop such a system both SAS and InSAS processing techniques must be understood, thus a basic introduction to SAS is now presented.

1.1 SYNTHETIC APERTURE SONAR

This section provides a basic description of the sonar hardware and how it interacts with the underwater scene, as well as detailing some of the signal processing algorithms used to generate SAS images from the received echos.

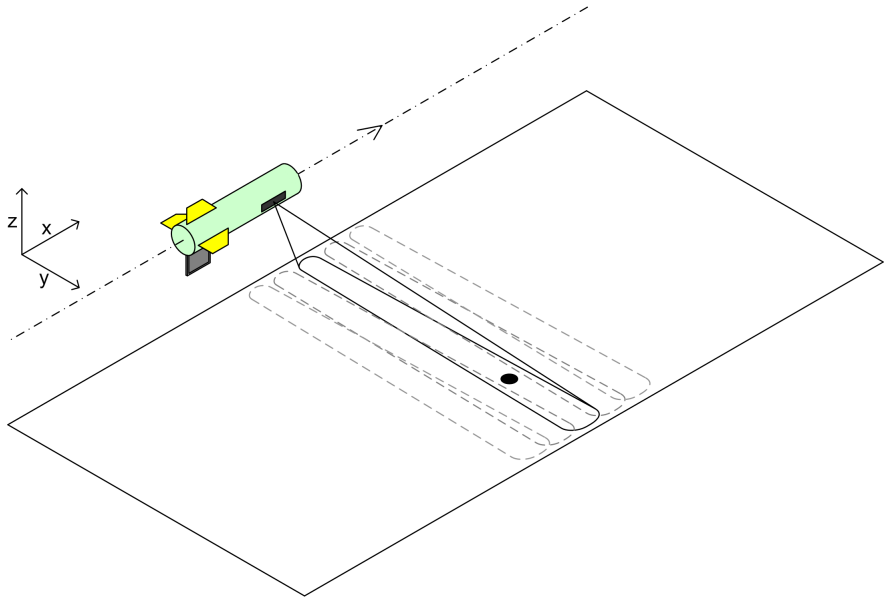
1.1.1 Towfish

SAS systems can be split into two categories: towed and non-towed. A towed system has the advantage that navigation on a large scale can be handled by the operators of the tow vessel. The towfish must still record micro-navigational data to allow for minor variations in the along-track path to be accounted for when assembling the SAS image [4]. A non-towed system is typically implemented on an underwater autonomous vehicle (UAV). This complicates the problem, as the system is also responsible for the navigation of the UAV and data can only be transmitted back to the operator's vessel when the UAV is on the surface. However, the ability of non-towed systems to image dangerous waters without risking human life makes them more attractive in military applications, while towed systems are typically more prevalent in research and commercial applications. A rail mounted system is another type of non-towed sonar. While it has the advantage of a high level of control of the sonar's path, it is not portable, so its use is purely restricted to research.

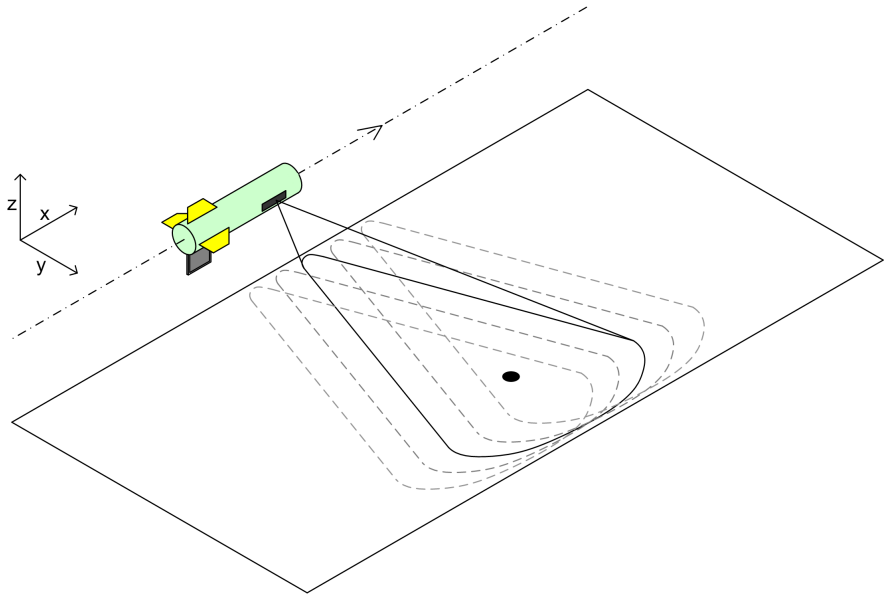
The KiwiSAS-IV is a towed design featuring a neutrally buoyant towfish that comprises a transmitter, a 3x3 receiver array and the associated electronics for each. Data is stored on a 250 GB hard disk for post-processing and the towfish communicates with a laptop on the tow vessel over a 100 Mbps Ethernet link. This is used for monitoring some of the raw data channels as well as sending configuration information to the towfish.

1.1.2 Operation

Figure 1.1 shows the differences between a traditional sidescan sonar and a SAS system. The sidescan sonar uses a transmitter with a narrow beam width so that only a small footprint is isonified. The echoes received can then be assumed to be from scatterers lying in that footprint. The beam width increases with range, so the resolution deteriorates in the across-track dimension. The SAS system uses a wide-beam transmitter to ensure that each scatterer appears in multiple pings, and crucially, is designed such that all pings have coherent phase. By using the phase information in each ping, adjacent pings can be combined to form the synthetic aperture. As the beam width increases with range, more information is available about scatterers at long ranges as they appear in more pings than scatterers where the footprint is narrower. This allows images to be reconstructed with range-independent resolution.



(a) Narrow-beam sidescan sonar.



(b) Wide-beam synthetic aperture sonar.

Figure 1.1 Comparison of a traditional sidescan sonar and a synthetic aperture sonar. In (a) the scatterer on the seafloor only appears in one ping, while in (b) it appears in 5 adjacent pings, allowing a synthetic aperture to be constructed.

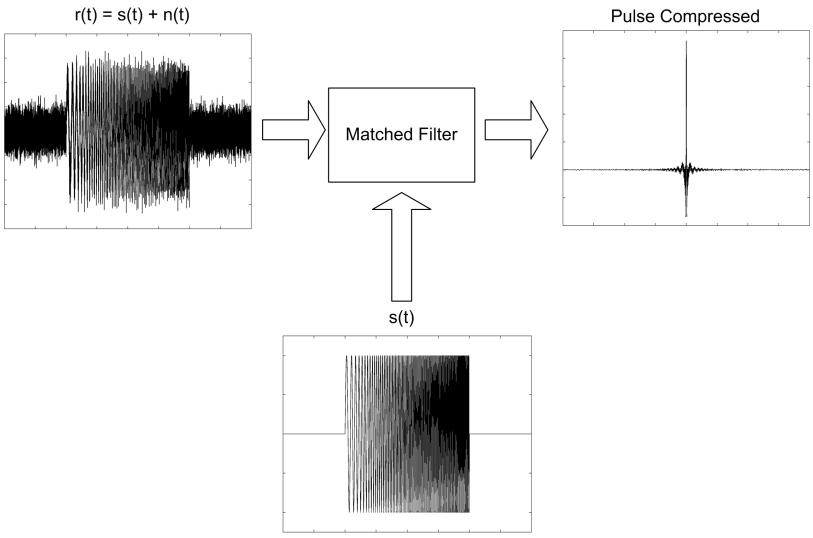


Figure 1.2 Pulse compression is achieved by correlating the received signal with a stored copy of the transmitted signal in a matched filter arrangement. This allows a high level of timing precision to be achieved with a long duration signal.

The transmitted signal is comprised of two FM chirps, the first from 20 kHz to 40 kHz and the second from 90 kHz to 110 kHz. In theory the low frequency chirp should offer better penetration of mud on the seafloor while the high frequency chirp should give better accuracy, although sediment penetration is heavily dependent on ripples in the seafloor [5]. The transducers used in the KiwiSAS-IV were designed to be used at 20–40 kHz but were found to work well in the 90–100 kHz band as well. The main reason for transmitting a chirp signal is to increase the SNR of the received echoes. With a constant frequency signal a short pulse is desirable as the SNR is governed by the timing precision. However, a short pulse requires impractically high instantaneous power to effectively isonify the seafloor. When a chirp is used, the returning signal can be correlated with the transmitted signal in a matched filter as it is received, giving an impulse-like waveform when the received signal matches the transmitted [6]. A relatively long pulse can then be used to deliver an appropriate amount of energy with a low instantaneous power, while still preserving a high level of timing precision. This process is called pulse compression, and is illustrated in Figure 1.2. Pulse compression has the advantage that the width of the impulse generated depends only on the bandwidth of the chirp, allowing the SNR to be increased by transmitting over a longer time.

Once the image has been compressed in the across-track direction, the synthetic aperture can be constructed by combining the individual pings into one image, a process usually referred to as azimuth compression. There are numerous methods to achieve this, including correlation, back-projection, range-doppler, wavenumber, and chirp-scaling. These methods will not be discussed here, for a more detailed analysis the

reader should consult [7] or [8].

This concludes the discussion of SAS. If more detail is required, the subject is discussed in detail in [2], [7], [8], and [9].

1.2 ASSUMED KNOWLEDGE

It is assumed that the reader has some knowledge of the basic operation of a SAS system; the explanation provided in this chapter should suffice, as in-depth understanding is not necessary to comprehend the operation of the system. The reader should also have some knowledge of digital signal processing techniques, especially their application in embedded systems. A basic understanding of embedded C programming and fixed-point number representation would be beneficial. The Q notation is used to describe fixed-point numbers at various points in the thesis; for an explanation of the notation the reader should consult [10].

1.3 THESIS OVERVIEW

Chapter 1 – An introduction to the SAS problem and a brief description of the sonar system is presented.

Chapter 2 – A discussion of the bathymetric problem that builds on the ideas introduced in the previous chapter. The mathematics describing the problem are presented as well as MATLAB implementation details. A brief introduction to the FPGA architecture is then given.

Chapter 3 – Discusses the System-on-Chip approach to embedded FPGA programming, comparing open-source and proprietary design options. A comparison with traditional embedded systems design is also presented.

Chapter 4 – This chapter deals with the implementation of the system in detail, covering both the device and host sides.

Chapter 5 – The results of the system are presented and discussed. The performance of each component of the design is analysed and possible improvements are suggested.

Chapter 6 – Conclusions are drawn and the improvements that need to be made to the system for full functionality are detailed.

Chapter 2

BACKGROUND

As described briefly in Chapter 1, SAS provides a 2D map of the reflectivity of objects on the seafloor. To form a 3D representation of the seafloor, a process called bathymetry is used. The chief concern of this thesis is to develop an implementation to accelerate this process, so this chapter describes the bathymetry problem in detail. The mathematical equations describing the problem are presented and compared with other applications that use similar methods and finally the FPGA architecture and its suitability for this application are examined.

2.1 INSAS

Bathymetric (or interferometric) SAS (InSAS) is a process that allows a 3D map of the seafloor to be constructed. This can take place any time after the pulse compression stage but typically occurs after azimuth compression in a SAS system. Much better results can be acheived after azimuth compression since the interferometry routine relies on a narrow beam-width (or narrow synthesised beam-width) to generate accurate results. The implementation presented here can be used in any type of sidescan sonar however, so azimuth compression is certainly not a prerequisite.

A typical sidescan bathymetry system utilises at least two vertically separated receivers as illustrated in Figure 2.1. The system aims to calculate the depth of the seafloor by estimating the angle between the signals received on each transducer. This is possible because for any given scatterer below the towfish, the echo will be picked up by the lower transducer before the upper. Once the angle of the incoming echoes and their ranges is known the height of the seafloor can be found by trigonometry. Finding the time delay between the incoming echoes is therefore the crux of the bathymetry problem. An example for a single scatterer is now given.

2.1.1 Single scatterer

Consider the system shown in Figure 2.1. If the transmitter is located at $(x_t,0,z_t)$, the scatterer at $(x_s,0,z_s)$ and the two receivers at $(x_{r_1},0,z_{r_1})$ and $(x_{r_2},0,z_{r_2})$ then each

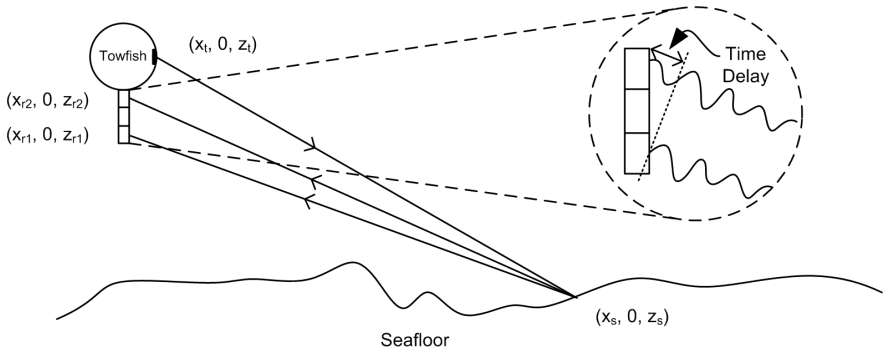


Figure 2.1 Geometry of an InSAS system for a single scatterer.

receiver will pick up a single echo of the transmitted ping, at the time given by the following equation:

$$t_{r_n} = \frac{\sqrt{(x_s - x_t)^2 + (z_s - z_t)^2} + \sqrt{(x_s - x_{r_n})^2 + (z_s - z_{r_n})^2}}{c} \tag{2.1}$$

where typically the speed of sound in water c is

$$c = 1500\,\mathrm{m/s} \tag{2.2}$$

After pulse compression, the two signals will look similar to those shown in Figure 2.2. The time delay can now be found by maximising the cross-correlation of the two signals, as this will clearly give the delay τ for which the two signals are aligned.

$$\tau = t_{r_2} - t_{r_1} \tag{2.3}$$

Once this delay is known, finding the height of the seafloor is simply an exercise in geometry, and is given by

$$\hat{z_s} = \frac{-r\tau c}{\mathrm{BL}} + z_0, \tag{2.4}$$

where $\hat{z_s}$ is the estimated height, r is the slant range, BL is the baseline — the distance between the phase centres of the two transmitter-receiver pairs, and z_0 is the distance from the surface to the midpoint between the two receivers. Further detail on the derivation of (2.4) can be found in [7].

Of course, in practice the scene will contain more than one scatterer, so the signals being processed will be the sums of many pulse compressed echoes. The height (and hence the time delay) will be different at every resolution cell along the ping, so evidently the time delay must be found by correlation for each sample in the ping. This is achieved using a sliding correlation window to calculate the delay at each sample.

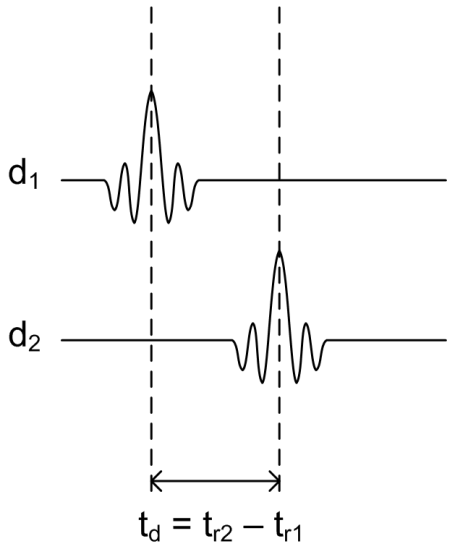


Figure 2.2 Pulse compressed signals for a single scatterer showing a time delay between the vertically separated receivers.

2.2 MATHEMATICAL MODEL

Now that the bathymetric problem has been described pictorially, the mathematics required to solve the time delay problem are presented. To find the height z_s at a point x_s, the correlation between the two signals should be maximised over the range of expected heights, giving the following equations, taken from [11].

$$\hat{z_s}(x_s) = \arg\max_{z_s} \left| \int_{-\infty}^{\infty} \chi_{12}(t, t_{r_1}, t_{r_2}) \right| \mathrm{d}t \tag{2.5}$$

where

$$\chi_{12}(t, t_{r_1}, t_{r_2}) = d_2(t - t_{r_2}) d_1^*(t - t_{r_1}) \times \mathrm{rect}\left(\frac{t - t_{r_1}}{T}\right) \mathrm{rect}\left(\frac{t - t_{r_2}}{T}\right) \tag{2.6}$$

and where d_1 and d_2 are the complex baseband signals received on hydrophones 1 and 2, and T is the extent of the correlation window.

Assuming a single scatterer, the problem is simplified to finding a time delay between the two signals as described above. This can be extended to multiple scatterers on the condition that there is only one scatterer per resolution cell. Equations (2.5) and (2.6) are now

$$\widehat{\Delta\tau}(t, \tau) = \arg\max_{\Delta\tau} \left| \int_{-\infty}^{\infty} \chi_{12}(t, \Delta t, \tau) \mathrm{d}t \right| \tag{2.7}$$

where

$$\chi_{12}(t, \Delta t, \tau) = d_2(t + \Delta\tau) d_1^*(t) \times \mathrm{rect}\left(\frac{t - \tau}{T}\right) \mathrm{rect}\left(\frac{t - \tau - \Delta\tau}{T}\right) \tag{2.8}$$

To provide meaningful results, the correlation result should be normalised to give a measure of the coherence, as (2.7) will give a result regardless of the actual level of correlation between the two signals. The normalised correlation coefficient is given by the standard form

$$\rho_{d_1 d_2} = \frac{\mathrm{E}\{(d_1 - \overline{d}_1)(d_2 - \overline{d}_2)^*\}}{\sqrt{\mathrm{E}\{\left|d_1 - \overline{d}_1\right|^2\}\,\mathrm{E}\{\left|d_2 - \overline{d}_2\right|^2\}}}. \tag{2.9}$$

In this application the assumption is made that the two signals are zero mean, so normalised covariance and correlation are equivalent. The correlation coefficient is therefore calculated as

$$\rho_{d_1 d_2}(\tau) = \frac{R_{d_2 d_1}(\tau)}{\sqrt{R_{d_1}(0) R_{d_2}(0)}} \tag{2.10}$$

where

$$R_{d_1} = \mathrm{E}\{|d_1(t)|^2\} \tag{2.11}$$

$$R_{d_2} = \mathrm{E}\{|d_2(t)|^2\} \tag{2.12}$$

and

$$R_{d_2 d_1}(\tau) = \mathrm{E}\{d_2(t+\tau) d_1^*(t)\}. \tag{2.13}$$

If d_1 and d_2 are wide-sense stationary and ergodic, the cross-correlation can be estimated using the time average formula

$$R_{d_2 d_1}(\tau) = \lim_{T \to \infty} \frac{1}{T} \int_{-\infty}^{\infty} d_{T_2}(t+\tau) d_{T_1}^*(t) \mathrm{d}t, \tag{2.14}$$

where d_{T_1} and d_{T_2} are windowed versions of the signals,

$$d_{T_1} = d_1(t) \mathrm{rect}\left(\frac{t}{T}\right) \tag{2.15}$$

$$d_{T_2} = d_2(t) \mathrm{rect}\left(\frac{t}{T}\right). \tag{2.16}$$

For a finite window length, the cross correlation is approximated by

$$\hat{R}_{d_2 d_1}(\tau) = \frac{1}{T} \int_{-\infty}^{\infty} d_{T_2}(t+\tau) d_{T_1}^*(t) \mathrm{d}t \tag{2.17}$$

$$= \frac{1}{T} \int_{-\infty}^{\infty} d_2(t+\tau) d_1^*(t) \mathrm{rect}\left(\frac{t}{T}\right) \mathrm{rect}\left(\frac{t-\tau}{T}\right) \mathrm{d}t. \tag{2.18}$$

The time delay estimate $\hat{\tau}$ is then given by

$$\hat{\tau} = \arg\max_{\tau} \{\hat{R}_{d_2 d_1}(\tau)\} \tag{2.19}$$

which is equivalent to (2.7). The correlation in (2.18) can be split into two regions

$$\hat{R}_{d_2 d_1}(\tau) = \begin{cases} \frac{1}{T}\int_{\tau-T/2}^{T/2} d_2(t+\tau)d_1^*(t)\mathrm{d}t, & \tau > 0, \\ \frac{1}{T}\int_{-T/2}^{\tau+T/2} d_2(t+\tau)d_1^*(t)\mathrm{d}t, & \tau \leq 0, \end{cases} \tag{2.20}$$

and the case $\tau \leq 0$ can be discarded if the seafloor is assumed to be below the receiver. If d_1 and d_2 are sampled with period Δt such that

$$d_n[m] = d_n(m\Delta t), \tag{2.21}$$

for a given window length M, the cross-correlation is described by

$$c[q] = \sum_{m=0}^{M-1} d_1[m]d_2^*[m+q], \tag{2.22}$$

where

$$d_1[m] = x_1[m] + jy_1[m] \tag{2.23}$$
$$d_2[m] = x_2[m] + jy_2[m]. \tag{2.24}$$

and where q varies over the range of delays being computed. Substituting (2.23) and (2.24) into (2.22) gives

$$c[q] = \sum_{m=0}^{M-1} (x_1[m] + jy_1[m])\,(x_2[m+q] - jy_2[m+q]) \tag{2.25}$$
$$= \sum_{m=0}^{M-1} x_1[m]x_2[m+q] + y_1[m]y_2[m+q] + j\sum_{m=0}^{M-1} y_1[m]x_2[m+q] - x_1[m]y_2[m+q]. \tag{2.26}$$

Equation (2.26) gives the correlation result for a certain lag, q. which is then varied over the expected range of time delays. Finding the maximum in magnitude of these correlation results gives a rough estimate of the delay between the two signals as a discrete number of samples. The time delay can then be more accurately estimated through interpolation methods.

2.2.1 Time delay estimation accuracy

Because of the long across-track distances in SAS systems, a small error in the time delay estimation translates to a large error in height estimation. It is therefore desirable to obtain as accurate an estimation as possible. The Cramér-Rao Lower Bound (CRLB) defines the minimum variance possible for a given estimation scheme, and thus provides

a useful measure of the accuracy. The general form of the CRLB for a delay estimate $\hat{D}$ is defined in [12] as

$$\mathrm{Var}\left[\hat{D}\right] \geq \left[2T\int_0^\infty (2\pi f)^2 \frac{|\gamma_{12}(f)|^2}{\left[1-|\gamma_{12}(f)|^2\right]}\mathrm{d}f\right]^{-1} \tag{2.27}$$

where $\gamma_{12}(f)$ is the spectral coherence. Knapp and Carter [12] showed that a maximum likelihood estimator would achieve this lower bound when the signal was weighted to favour frequencies with high coherence or low noise. A cross-correlator with a flat frequency weighting will not achieve the CRLB unless the signal and noise power spectra match each other. This is discussed in more detail in [13].

Typical correlation results from each estimation scheme are shown in Figure 2.3. The highest accuracy estimate is found when the correlation is carried out on the real data. More commonly the correlation is performed on complex baseband data, giving poor accuracy when the delay is estimated from the magnitude of the correlation. To determine the reasons for this, the chirp signal itself must first be examined. Consider the ideal case

$$S(f) = \mathrm{rect}\left(\frac{f-f_c}{B}\right) \tag{2.28}$$

the center frequency is

$$f_0 = \frac{\int_0^\infty f|S(f)|^2\mathrm{d}f}{\int_0^\infty |S(f)|^2\mathrm{d}f} \tag{2.29}$$

$$= \frac{\int_{f_c-B/2}^{f_c+B/2} f\mathrm{d}f}{\int_{f_c-B/2}^{f_c+B/2}\mathrm{d}f} \tag{2.30}$$

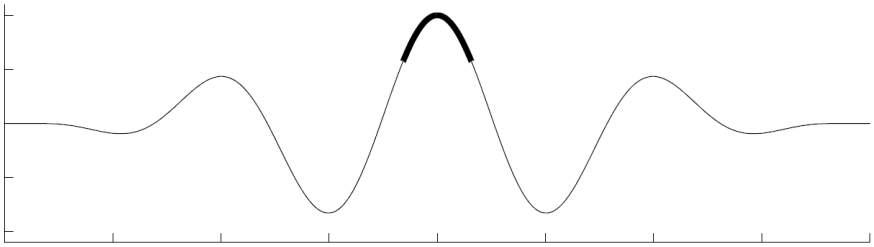
$$= f_c \tag{2.31}$$

the rms bandwidth is

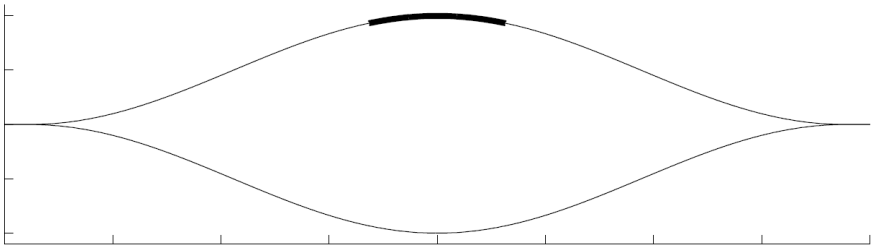
$$\beta^2 = \frac{\int_0^\infty (f-f_0^2)^2|S(f)|^2\mathrm{d}f}{\int_0^\infty |S(f)|^2\mathrm{d}f} \tag{2.32}$$

$$\beta = \sqrt{\frac{\int_{f_c-B/2}^{f_c+B/2}(f-f_c^2)^2\mathrm{d}f}{\int_{f_c-B/2}^{f_c+B/2}\mathrm{d}f}} \tag{2.33}$$

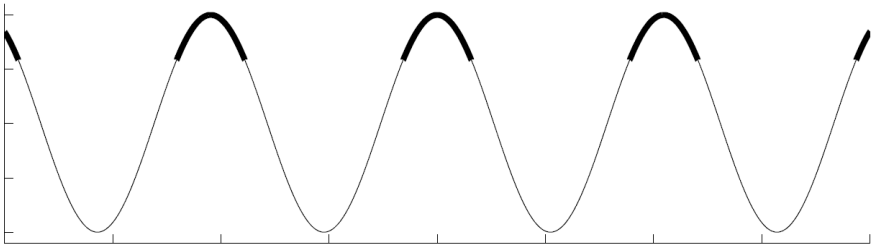
$$= \frac{B}{\sqrt{12}} \tag{2.34}$$



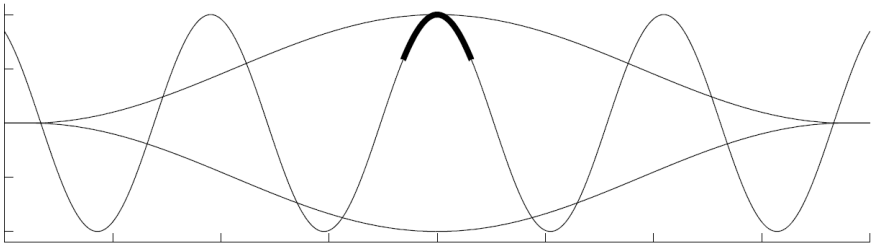
(a) Modulated real correlation.



(b) Magnitude-only complex correlation.



(c) Phase-only complex correlation.



(d) Magnitude then phase (quasi-narrowband) complex correlation.

Figure 2.3 Estimation schemes for finding the peak of a correlation, with bold lines showing curvature around estimate. Reproduced from [7]. Higher curvature indicates a lower variance estimate. The most accurate estimate is generated using modulated real correlation (a) but the high sampling rate required is computationally intensive. Similar performance can be achieved using phase-only complex correlation (c) but results are subject to 2π ambiguity. The coarse peak estimate provided by magnitude-only complex correlation (b) can be used to eliminate the ambiguity in a quasi-narrowband approach (d).

and the root of the second moment is

$$\beta_0^2 = \beta^2 + f_0^2 \tag{2.35}$$

$$\beta_0 = \sqrt{\beta^2 + f_0^2} \tag{2.36}$$

$$= \sqrt{\frac{B^2}{12} + f_c^2} \tag{2.37}$$

The CRLB for a system operating on real data is derived in [13, pp. 44–45] from (2.27) as

$$\mathrm{Var}\left[\hat{\tau}\right] \geq \frac{1}{(2\pi f_c)^2} \frac{1}{TB} \frac{1}{1 + \frac{B^2}{12 f_c^2}} \left[\frac{1}{\mathrm{SNR}} + \frac{1}{2\mathrm{SNR}^2}\right] \tag{2.38}$$

By substituting (2.37), this can be rewritten as

$$\mathrm{Var}\left[\hat{\tau}\right] \geq \frac{1}{4\pi^2} \frac{1}{TB} \frac{1}{\beta_0^2} \left[\frac{1}{\mathrm{SNR}} + \frac{1}{2\mathrm{SNR}^2}\right] \tag{2.39}$$

As mentioned above, the correlation is often performed on complex baseband data to reduce the sampling rate and hence the computational load of the processing. In this case the output of the cross-correlation is the envelope of the result that would be obtained using real data, which gives a much wider peak and hence a lower level of accuracy. The CRLB of the magnitude of complex baseband correlation is found by setting the centre frequency f_c to zero, so (2.39) becomes

$$\mathrm{Var}\left[\hat{\tau}_c\right] \geq \frac{1}{4\pi^2} \frac{1}{TB} \frac{1}{\beta^2} \left[\frac{1}{\mathrm{SNR}} + \frac{1}{2\mathrm{SNR}^2}\right] \tag{2.40}$$

A finer estimate can be obtained from the complex baseband correlation by finding the peak in the phase of the correlation result, but this suffers from 2π ambiguity. This can be resolved by choosing the result closest to the coarse estimate, sometimes referred to as a quasi-narrowband approach. The accuracy in this case is greatly improved over the complex baseband magnitude result but slightly less than the real data method because of the triangular windowing effect of the basebanding operation. The CRLB is found in this case by setting the bandwith B to zero in (2.34) (the time-bandwidth product is not affected) giving

$$\mathrm{Var}\left[\hat{\tau}_f\right] \geq \frac{1}{4\pi^2} \frac{1}{TB} \frac{1}{f_0^2} \left[\frac{1}{\mathrm{SNR}} + \frac{1}{2\mathrm{SNR}^2}\right] \tag{2.41}$$

Thus, the result given by the CRLB is dependent on (2.35). Clearly, the lowest variance occurs when the correlation is performed on real data, as (2.35) is maximised. The KiwiSAS system typically uses a chirp with a bandwidth of 20 kHz, which will give an rms bandwidth of 5.77 kHz by (2.34) assuming an ideal spectrum. As the centre frequency is increased the rms bandwidth will have proportionally less effect on

β_0 than f_0 and the quasi-narrowband approach will provide a better estimate of the actual delay.

2.2.2 Algorithm summary

Solving the bathymetric problem can thus be broken down into a set of steps.

1. Calculate correlation of input signals with sliding window. Gives magnitude of correlation estimate to nearest sample.
2. Interpolate magnitude of correlation estimate to improve accuracy.
3. Normalise correlation result.
4. Calculate phase of correlation estimate and take value closest to magnitude of correlation result.
5. Construct height map from delay estimates.

Step 1 is clearly the most computationally intensive of these, requiring a large number of multiply and accumulate operations.

2.3 SIMILAR APPLICATIONS

Cross correlation methods are also used in other areas such as 802.11a packet detection [14], mobile phone positioning [15], and of course synthetic aperture radar (SAR) [16], a field that shares many common elements with SAS. This section examines applications in these fields to ascertain the similarities and evaluate which elements of the design can be exported to a sonar environment.

2.3.1 802.11a packet detection

The packet detection scheme described in [14] uses a correlation/covariance calculation to detect pilot symbols in an 802.11a packet. The start of a packet contains a 16-sample symbol repeated 10 times, followed by a guard interval and a 64-sample symbol repeated twice as shown in Figure 2.4. The detection algorithm aims to find the start of a packet by cross correlating the received signal with itself delayed by the length of a symbol. If the output of this is above a certain threshold, a symbol is said to be detected. For a packet to be detected, the algorithm requires that the output of the correlation is above the threshold for 10 symbols then below the threshold during the guard interval. The time delay is then increased to 64 samples, and the output must rise above the threshold again for the final two symbols.

Of particular interest is the integrate-and-dump scheme used to implement the correlation. As only one sample changes between correlations, it is not necessary to

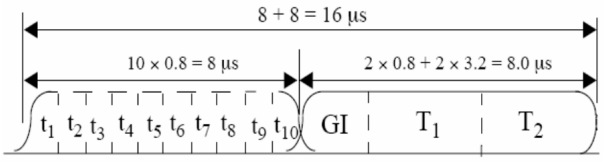


Figure 2.4 802.11a pilot symbol, reproduced from [14].

calculate the entire correlation each time. The product terms are simply kept in a RAM buffer, and the correlation sum is updated at each iteration by subtracting the oldest product and adding the new product. The same principle can be applied in the bathymetry problem when normalising the correlation.

2.3.2 Mobile phone positioning

Many methods are currently available for obtaining the position of mobile phones; two that have the most in common with bathymetric methods are the Time Of Arrival (TOA) and Time Delay Of Arrival (TDOA) approaches. Both methods require at least three base stations.

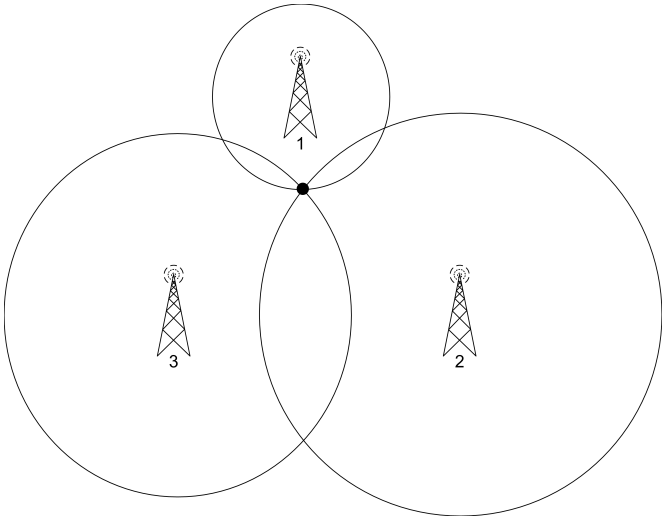
TOA uses the time of flight from the phone to two different base stations to provide an ambiguous fix on the phone's position in two dimensions, and a third to remove the ambiguity. This is illustrated in Figure 2.5. In the bathymetry system, the location of the seafloor is assumed to exist in a particular quadrant, so only two receivers are required.

The TDOA approach uses the time difference between two base stations for a given received signal to define a hyperbolic curve where the phone could feasibly exist. This is repeated for another pair of base stations and the intersection between the two hyperbole gives the location of the phone. Estimating the TDOA requires cross-correlation of the two received signals in a method similar to the bathymetric problem [17]. TDOA techniques are also used in the location of seismic events [18].

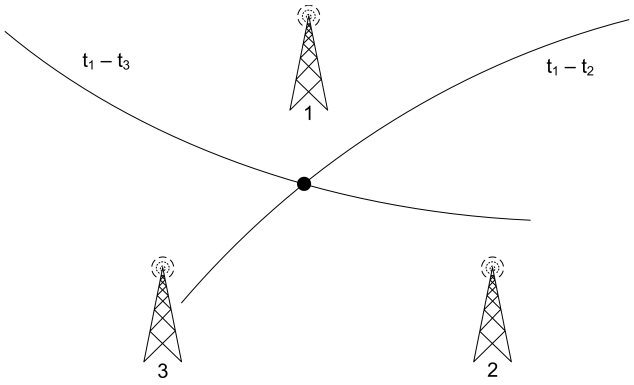
2.3.3 Synthetic aperture radar

Synthetic aperture radar (SAR) is a field very closely related to SAS. In a typical SAR system, electromagnetic waves are transmitted from an aeroplane or satellite and the reflections from the Earth recorded in a similar method to acoustic methods in SAS. Likewise, multiple small aperture signals are combined into a large synthetic aperture. The CARABAS system [16] is one such system, comprising hollow inflatable aerials towed behind a small jet aircraft.

The interferometric SAR case is analogous to the InSAS case; vertically separated receivers are used to form two coherent images and the phase difference between the two gives an estimate of the height. However, differences in the geometries of the two



(a) TOA mobile phone positioning.



(b) TDOA mobile phone positioning.

Figure 2.5 Comparison of techniques for the positioning of mobile phones.

prevent the direct application of SAR algorithms to the bathymetric SAS problem [7, pp. 30–31].

2.4 FPGA ARCHITECTURE

A Field Programmable Gate Array (FPGA) is a large configurable IC (integrated circuit) chip used in applications such as digital signal processing, telecommunications and instrumentation. FPGAs originated in the mid 1980s and were originally used as "glue-logic" in PCB designs with many discrete digital components. As CMOS technology has shrunk in size, the capacity of FPGAs has increased, allowing more complex designs to be implemented to the point now where entire systems can be implemented within the FPGA [19]. This methodology is called System-on-Chip (SoC) and is discussed in the next chapter. In this section, FPGA architectures and features are discussed and compared with conventional processors for their suitability to the bathymetry application.

2.4.1 Architecture

The basic element of the FPGA architecture is the configurable logic block (CLB), consisting of multiple inputs, a lookup table (LUT) and a D flip-flop to register the output as shown in Figure 2.6. The key part of this setup is the highly flexible LUT, which can basically be thought of as a small memory device. For example, a two-input LUT could be configured as a NAND gate by storing a 1 in the memory elements addressed by 00, 01 and 10 and a 0 in the element addressed by 11. In late-model FPGAS such as the Virtex-II Pro ^{TM}, the CLB consists of two four-input lookup tables and two registers as well as simple logic to combine the output of the LUTs, allowing much more complex logic functions to be implemented. LUTs of this size can also be easily configured as 16-bit shift registers or as memory in a distributed RAM model.

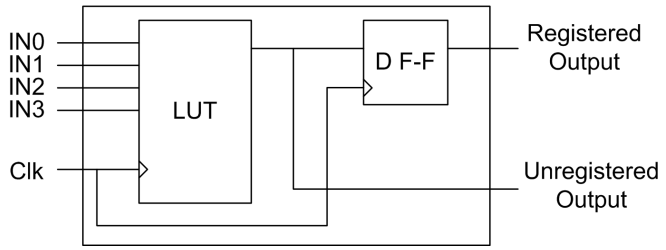


Figure 2.6 A simple complex logic block element.

The FPGA itself is made up of multiple CLBs in a "sea" of interconnect, as depicted in Figure 2.7. The Virtex-II Pro^{TM} 2P7-FG456 device that was used in this project contains approximately 11,000 CLBs. Interconnect wires run vertically and

horizontally, and programmable switches at interconnect crossings allow the CLBs to be connected together. Interconnections typically run in groups, and to reduce complexity the switches only allow each vertical interconnect to be connected with its partner on the horizontal interconnect. Signals that are sensitive to skew and jitter, such as the clock, are not routed on the standard interconnect but on a separate network designed to minimise timing errors [20].

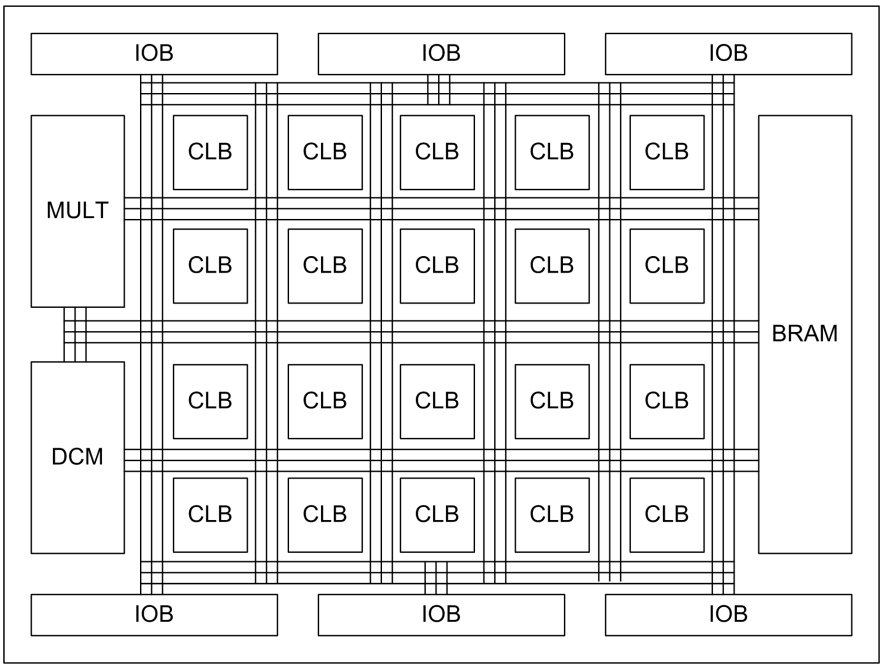


Figure 2.7 A block diagram of the internal structure of an FPGA.

Another significant feature in recent FPGAs is the non-programmable device area. This can include components such as digital clock managers (DCMs), block RAM (BRAM), multipliers, IO transceivers, and even microprocessor cores. These primitives are hard-wired into the silicon of the FPGA, allowing higher performance than synthesised components without using any logic area. This is especially important in DSP applications such as parallel FIR filters [21] that require a large number of multipliers. In the case of the DCM, a hard-wired component is the only solution that will meet the strict timing requirements for clock manipulation.

Of course, the multitude of resources within the FPGA must be able to interface somehow with the environment outside the chip, so FPGAs typically have extensive I/O capabilities, although the extent of these varies depending on the package. On the 2P7-FG456 device 248 I/O blocks are available to the user [22]. The logic driving each I/O block is shown in Figure 2.8 below. The three registers can either be edge-driven or configured as level sensitive latches, useful for input capture, and each I/O pin can be configured as in, out, or bi-directional. The Virtex-II Pro^{TM} I/O blocks support double data rate (DDR) applications with the dual register setup shown in

2.8, with each register driven by clocks 180^o out of phase with each other. I/O blocks are organised in banks, as shown in Figure 2.7, and each bank can be configured to a different I/O standard. A multitude of I/O standards are supported including LVCMOS, HSTL, SSTL, and GTL.

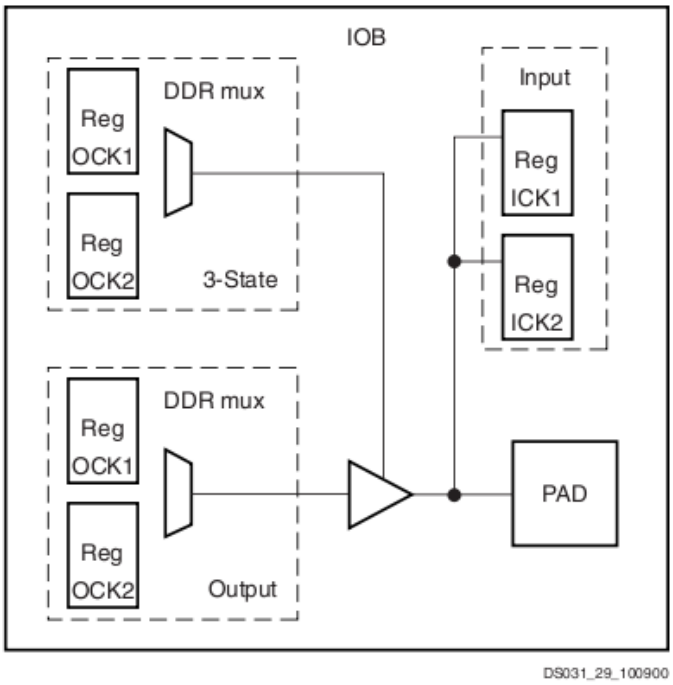


Figure 2.8 Structure of a Virtex-II Pro IO block, reproduced from [22].

2.4.2 Advantages

FPGAs allow the implementation of highly parallel designs. For example, a FIR filter implementation on a conventional DSP must fetch operands, multiply-and-accumulate, and store the result for each tap sequentially. With a large enough FPGA device, each tap can be computed in parallel, greatly increasing the speed of the filter.

Another advantage of the FPGA is that it effectively provides the designer with a "blank slate" in that the architecture of the system to be designed is completely open. This allows a highly optimised design that is tailored to the application, comparable to an ASIC design but without the high production costs.

This leads to the high level of flexibility inherent in the FPGA platform, indeed, FPGAs are often used in industry to prototype and test ASIC designs prior to production. Because the entire contents of the FPGA are configurable, the parts of a design that reside in an FPGA can be reconfigured at any stage in the design process. In SRAM-based FPGAs such as Xilinx and Altera devices, parts of the chip can even be reconfigured while the design is operating [23], allowing applications to dynamically

adapt to changing requirements.

2.4.3 Disadvantages

FPGAs also have disadvantages compared to traditional DSP architectures, the most prominent of which is the difficulty in programming them. The two most commonly used languages – VHDL and Verilog – are low-level and verbose, especially in the case of VHDL [24]. While low-level languages give the designer a lot of control over the implementation of the system, they also can result in large and complicated designs, making re-use of code more difficult. Additionally, some functionality will need to be created on an FPGA that is already available "out-of-the-box" in a typical DSP, like a single-cycle MAC for example. These complexities become more significant in the large scale designs that late-model FPGAs allow . SystemC [25] has been developed in an effort to provide a high-level language for FPGA programming, but its use has yet to become widespread.

Compared to a high-end DSP, an FPGA is typically more expensive due to its increased complexity. This increased cost has prevented the widespread adoption of FPGA technology in industry, as the advantages provided are often not worth the extra expense, especially if an operational DSP/microprocessor design is already available.

FPGAs typically use more power than microcontroller designs, since power-saving features are more difficult to build in when the entire design space is flexible. Thus in portable applications where battery life is paramount, a DSP or ASIC will often be used in place of an FPGA.

2.4.4 Suitability

As this application is a prototype with no strict power requirements, the only disadvantage is the programming difficulty. The parallel processing power is a big attraction of the FPGA platform for the bathymetry application, since it will speed up the correlation algorithm as well as allowing multiple pings to be processed simultaneously. When weighed against these advantages the programming complexity is a minor concern, and can be alleviated to some extent by employing the SoC design approach discussed in the next chapter.

Chapter 3

SYSTEM ON CHIP

3.1 OVERVIEW

As CMOS technology shrinks further into the sub-100 nm range FPGAs are becoming more powerful, allowing the hardware designer to implement more components of digital systems inside the chip, in what is called a "System-on-a-Chip" (SoC) design [19]. A SoC design comprises the same components as a traditional embedded design, the difference being that they are all implemented as modules inside the FPGA. Typically these modules include a microprocessor, a small amount of RAM, I/O peripherals, and some form of hardware acceleration.

SoC systems offer many advantages over traditional embedded systems, the most significant being more flexibility, allowing the designer to tailor system components to their needs and combat microprocessor obsolescence by using soft cores that are easily upgradeable. Printed circuit board layouts are simplified due to the reduced number of components, and this in turn can lead to lower costs in some designs [26].

Various methods for SoC design are available, using either proprietary tools specifically targeted at SoC applications such as Xilinx's Embedded Development Kit (EDK) or an open source solution using freely available cores from repositories such as OpenCores [27]. The EDK approach was used in this project; this chapter discusses the elements of an EDK design and compares them with the open source alternative.

3.2 PROCESSORS

The heart of a SOC project is usually an embedded processor core. Xilinx offers three proprietary choices: PicoBlaze [28], MicroBlaze [29] and PowerPC 405 [30]. Numerous open source microcontrollers are also available from OpenCores, but the MicroBlaze and PowerPC have the advantage of being fully integrated into Xilinx's Embedded Development Kit (EDK) tool.

3.2.1 PicoBlaze

The PicoBlaze is a free 8-bit RISC soft microcontroller core, useful for replacing complex state machines. The architecture is shown in Figure 3.1. The 16-bit instruction bus connects to the output of a BRAM module. Generally, the small data and address buses make the PicoBlaze better suited to control applications than signal processing or data management. Programs are loaded into the PicoBlaze by setting the initialisation strings for the PicoBlaze's BRAM before the design is synthesised. The pBlazIDE [31] tool provides a means to compile PicoBlaze assembler code into BRAM initialisation strings and includes a basic simulator.

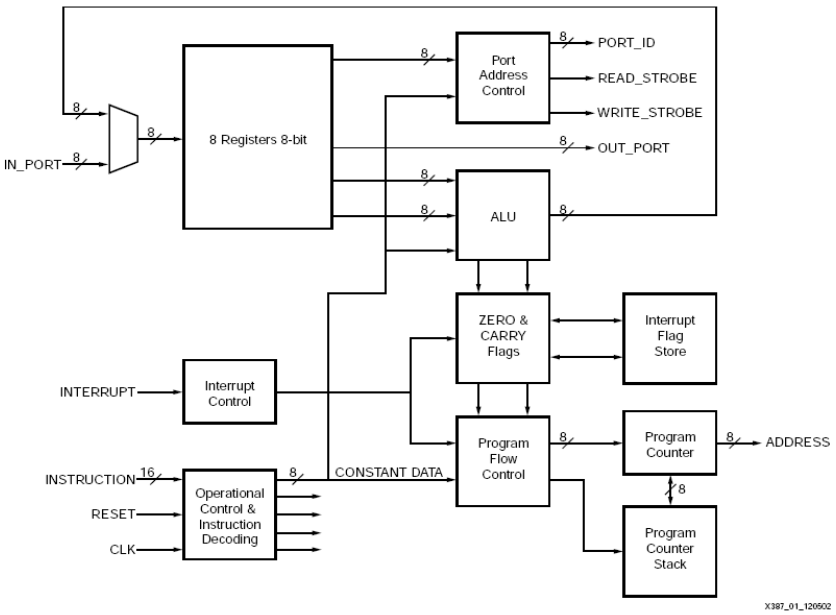


Figure 3.1 PicoBlaze microprocessor architecture, reproduced from [28].

3.2.2 MicroBlaze

The MicroBlaze is a 32-bit RISC, Harvard architecture soft core processor that is distributed with EDK. It is theoretically capable of 166 DMIPS (Dhrystone MIPS) with clock speeds up to 200 MHz. Data and instructions are stored in internal block RAM, which allows single cycle execution over the Local Memory Bus (LMB). The MicroBlaze core is highly customisable, allowing the user to add hardware or software debugging, a floating point unit, or a cache for On-Chip Peripheral Bus (OPB) data. It is also possible to speed up operations such as multiplication, division and barrel-shifting by adding hardware accelerated modules to the core as shown in Figure 3.2. Software development for the MicroBlaze is supported with GNU tools integrated into EDK to compile C code. The MicroBlaze was thus chosen for use in the project.

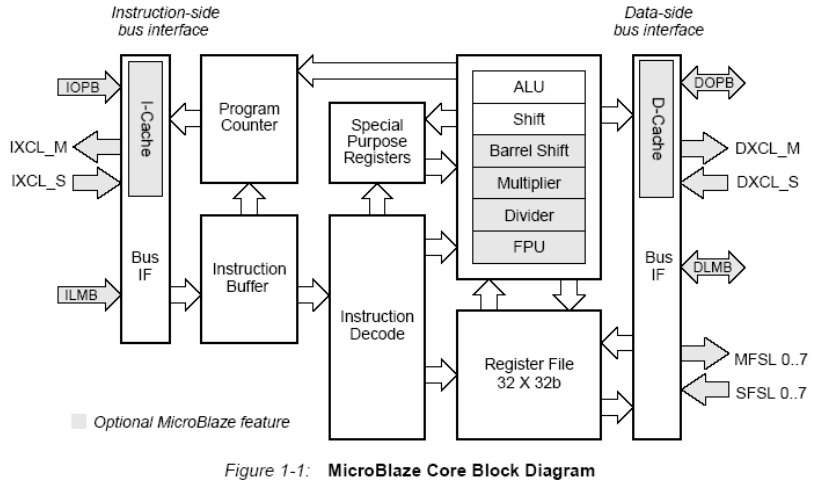


Figure 1-1: **MicroBlaze Core Block Diagram**

Figure 3.2 MicroBlaze processor architecture, reproduced from [29].

3.2.3 PowerPC 405

The PowerPC 405 core is also a 32-bit RISC processor similar to the MicroBlaze but capable of much higher performance. Additionally, it is hard-wired into the silicon of the FPGA, requiring no extra device area to implement. It also adds extra features such as a Memory Management Unit (MMU) allowing Real-Time Operating System (RTOS) implementation, hardware multiply and divide, and increased clock speeds of up to 400 MHz. The PowerPC 405 architecture is shown in Figure 3.3. The downside is that the PowerPC can only be implemented on the Virtex-II Pro^{TM} and Virtex-4^{TM} families of devices. For this reason although a PowerPC core is available on the Virtex-II Pro^{TM} device, the MicroBlaze has been used to improve portability across FPGA platforms.

3.2.4 Open source

A wide range of free microcontroller cores of varying complexity are available from OpenCores, ranging from simple 8-bit microcontrollers to high performance DSP units. The cores predominantly employ RISC architectures but other instruction set architectures are also available.

The most full-featured core is the OpenRISC 1000 series, a family of configurable 32 or 64-bit RISC microprocessors designed for medium to high performance networking and embedded applications. OpenRISC 1000 processors include an MMU and can be configured to include DSP, vector and FPU blocks. Functionally, the specifications are similar to the PowerPC 405 core, but as they are soft cores the maximum clock frequency is limited to around 160 MHz. The architecture of the OpenRISC 1200 is shown in Figure 3.4 below.

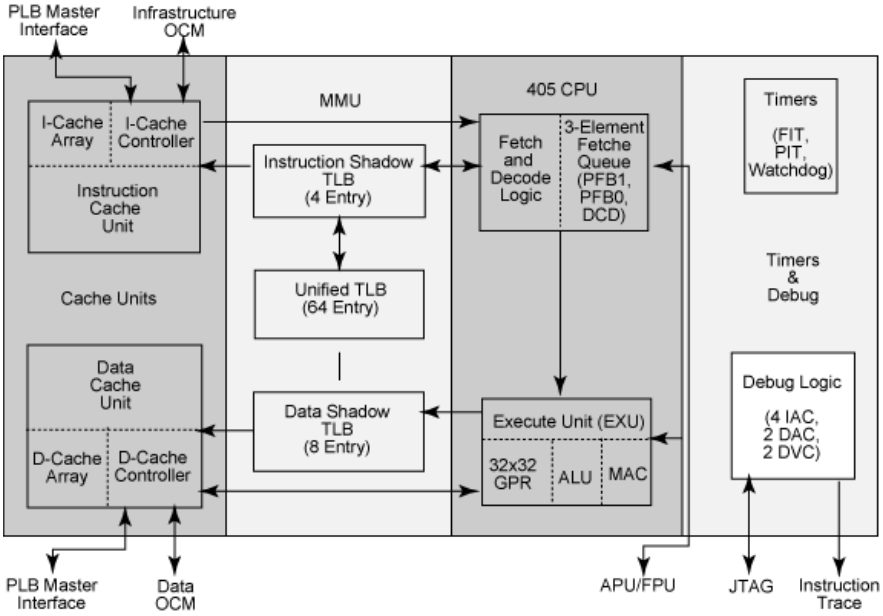


Figure 3.3 PowerPC 405 microprocessor architecture, reproduced from [32].

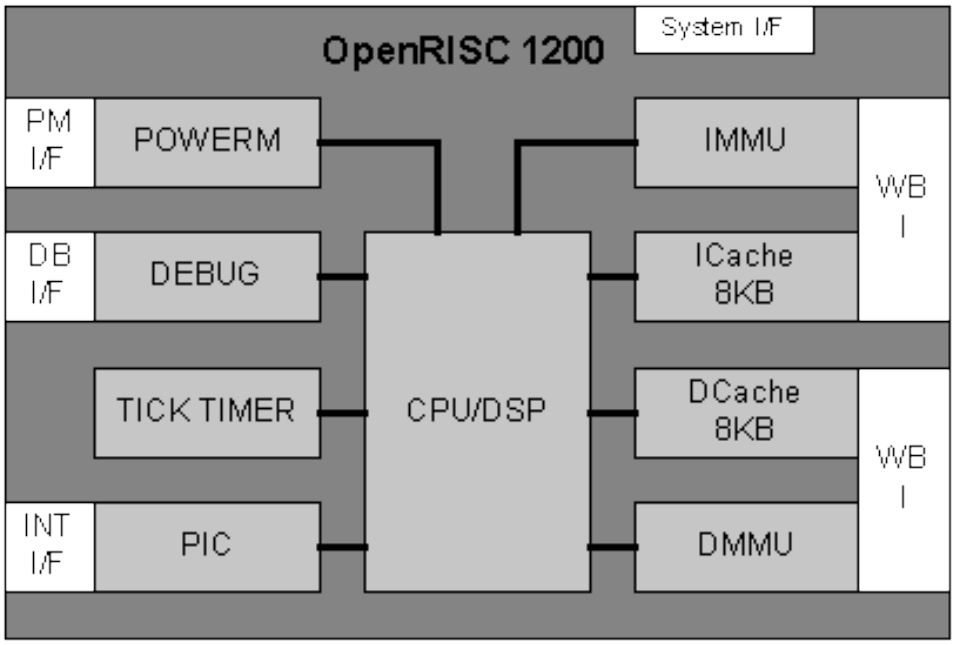


Figure 3.4 OpenRISC 1200 microprocessor architecture, reproduced from [33].

One of the principle advantages of the OpenRISC 1000 is the availability of a fully functional GNU toolchain for the processor, allowing the user to compile C code to run on the core. The lack of a C compiler for most of the open source processors is one of the major drawbacks to using an open source solution over a proprietary solution, so the OpenRISC goes a long way towards closing the gap.

3.3 BUS ARCHITECTURES

In larger SoC designs, an elegant method of connecting modules together is required, and to encourage re-use of modules it is desirable that the bus interfaces adhere to a standard. The PicoBlaze microcontroller has no set bus standard, since its bus interfaces consist only of an 8-bit data bus and an 8-bit address bus. MicroBlaze and PowerPC systems each have their own bus architectures with some common elements; open source systems typically use the Wishbone bus.

3.3.1 MicroBlaze

The MicroBlaze core communicates with other modules on the FPGA via three standardised buses. The Local Memory Bus (LMB), the On-board Peripheral Bus (OPB) and the Fast Simplex Link (FSL). A block diagram of the architecture is shown in Figure 3.5.

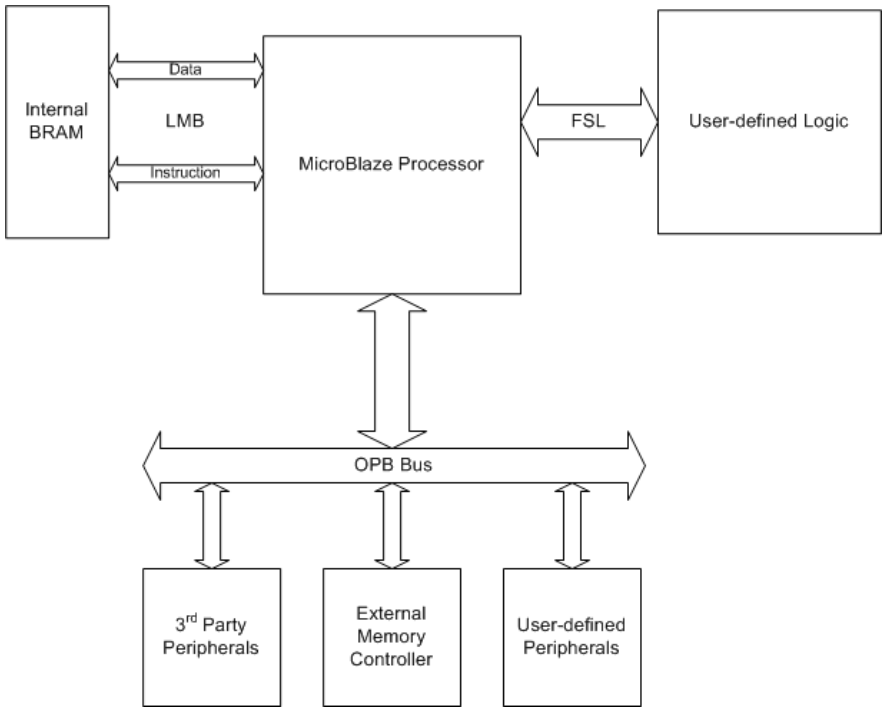


Figure 3.5 MicroBlaze processor architecture.

3.3.1.1 LMB

The LMB is a 32-bit single-master bus that provides the Harvard side instruction and data interfaces between the MicroBlaze and on-chip block RAM (BRAM), providing access to data and instructions. The MicroBlaze is always the bus master and communicates with local memory bus controllers on each side that each read and write to one port of the dual-port BRAM, allowing single-cycle access on the instruction side and two-cycle access on the data side. Accessing the data side takes longer because the data transfers can be bi-directional, while instruction transfers are always unidirectional, from the BRAM to the processor. The LMB controller is capable of interfacing to memory sizes from 8 to 64 kB, depending on the number of BRAM blocks used. The address size for the controller is specified when it is declared; the number of BRAM blocks required is then calculated upon synthesis.

3.3.1.2 OPB

The OPB is a multiple-master bus used for communications with peripherals such as external memory controllers, communication cores, and timers. The bus is 32 bits wide by default, but byte-enable signals can be used to transfer data of sizes 8, 16 and 32 bits. The MicroBlaze is always a master on the OPB, but other peripherals can also act as masters, as is the case with the 10/100 Ethernet MAC controller. The Ethernet controller has DMA functionality that allows it to read and write directly to the external memory controller, so in this scenario an OPB arbiter is employed to pass control of the bus between the MicroBlaze and the Ethernet core. The arbiter is automatically synthesised if the design requires it.

The ability to use DMA on the OPB is particularly useful as it is slower than the LMB and decreases in speed as more masters are attached to it. Accessing a peripheral on the OPB can theoretically be as quick as 2 cycles if the transfer is simply a register access without multiple master arbitration, but single external memory accesses are typically around 12 cycles. Allowing an IO peripheral to write directly to external memory lessens the effect of this, not only because the data is only transferred once, but also because the MicroBlaze can be working on a separate task during the data transfer.

If the program running on the MicroBlaze is particularly large, some sections may have to be loaded into external memory. This can drastically reduce system performance, as the access times for instructions and data are now significantly longer. Fortunately, the MicroBlaze configuration allows the user the option to enable data and instruction caches for memory on the OPB, which greatly improves performance; cache hits have similar latency to internal RAM accesses.

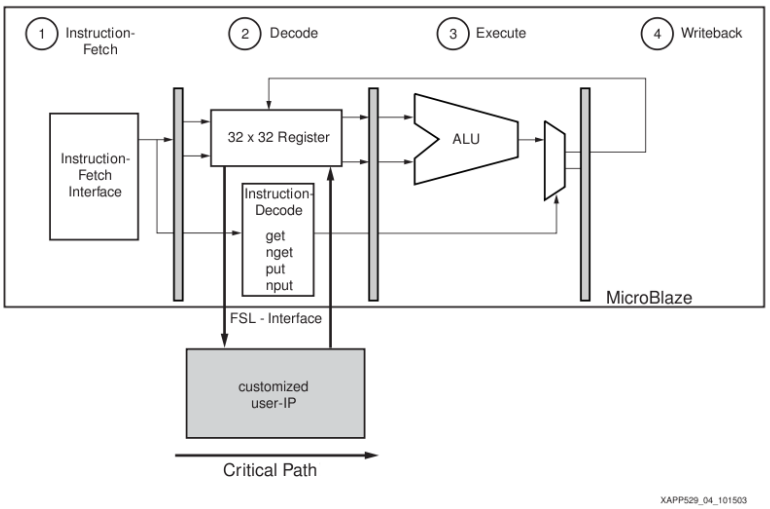


Figure 3.6 Position of FSL transactions in MicroBlaze datapath, reproduced from [34].

3.3.1.3 FSL

While the MicroBlaze is capable of performing complex signal processing tasks, the user may want to take advantage of the FPGA's parallel processing power by implementing more computationally intensive algorithms in hardware. The Fast Simplex Link (FSL) provides a high speed uni-directional asynchronous FIFO interface, useful for implementing a coprocessor in hardware or transferring data between MicroBlaze instances.

Transactions on the FSL are integrated into the pipeline as illustrated in Figure 3.6. The software interface is provided by the macros `put_fsl()` and `get_fsl()`, which can be executed as either blocking or non-blocking instructions. Blocking instructions check the status of the FSL FIFO before executing; a blocking write will stall the processor until there is space in the FIFO, and a blocking read will stall the processor until there is data to be read. In multi-threaded applications this can cause one thread to use a large amount of CPU time waiting for an instruction to unblock, so in these situations the non-blocking macros are used. After a non-blocking instruction, the MicroBlaze Status Register (MSR) must be checked manually to ensure that data was written successfully, or that valid data was read.

The interface in hardware at the other end of the FSL is very similar to a standard asynchronous FIFO interface. An incoming link has output data, data available, and read enable signals, as well as clock and reset signals. An optional clock input is provided to allow the user to run the coprocessor at a different speed to the rest of the system. An outgoing link has a similar interface, with input data, write enable, and full signals. The interface is shown in Figure 3.7 below. The control signal is a single bit channel that can be used to send an enable signal, or as an extra data bit.

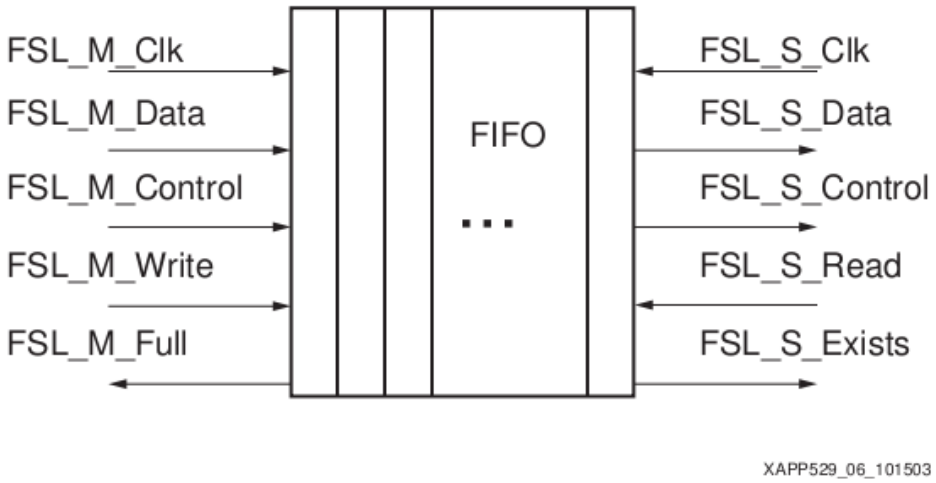


Figure 3.7 FSL FIFO interface, reproduced from [34].

3.3.2 PowerPC

The PowerPC 405 utilises the IBM CoreConnect standard architecture [35] exclusively for on-chip communications. This consists of the Processor Local Bus (PLB), the OPB described above, and the Device Control Register (DCR) Bus. Figure 3.8 shows the connections between buses.

3.3.2.1 PLB

The PLB is a high-performance bus used for communications between the PowerPC core, high-speed peripherals and hardware acceleration blocks. Address width is 32 bits and data width can be 32 or 64 bits. Separate buses are provided for read and write transactions, allowing overlapping read and write commands. This facilitates up to two transfers per cycle and transfers can occur in bursts of 16 to 64. Addressing is pipelined to reduce latency and the bus features built-in support for arbitration. Special DMA modes are available, including flyby and memory to memory.

3.3.2.2 OPB

The OPB described above is used for communication with low-speed IO peripherals and operates in almost exactly the same way that it does in a MicroBlaze system. The difference is that the OPB is bridged to the PLB allowing communications between modules on both buses.

3.3.2.3 DCR

One of the ways the PLB performance is increased is by removing configuration and status transfers from the bus. Since these transfers do not need to be high speed, they are executed on a slower bus, the Device Control Register (DCR) bus. All modules on the PLB are daisy-chained to the DCR bus, allowing configuration and status transfers to take place at the same time as high speed data transfers on the PLB.

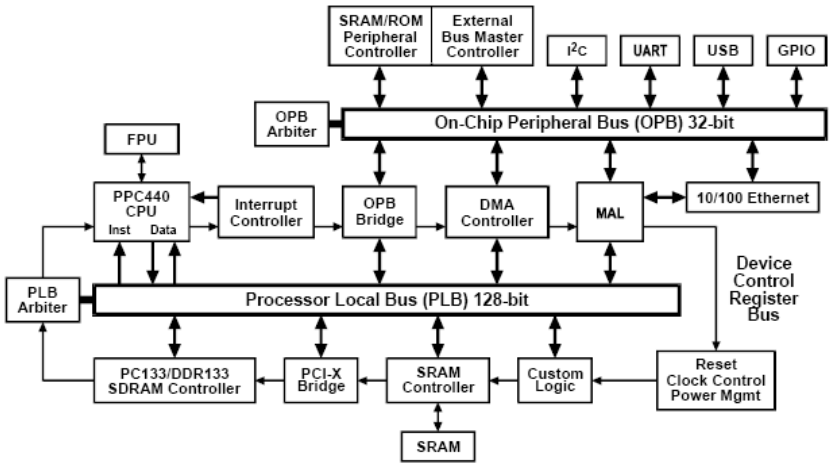


Figure 3.8 IBM CoreConnect architecture, reproduced from [35].

3.3.3 Wishbone

Wishbone [36] is a SoC bus specification developed by Silicore Corp., who transferred stewardship to the OpenCores community in 2002. It is free for use in the public domain and not subject to any licensing conditions. It is also the only interface standard used in OpenCores cores. A block diagram of a Wishbone system is shown in Figure 3.9 below.

The Wishbone interface is a simpler architecture than CoreConnect. One high speed, multiple-master bus connects all the cores in the design together, without the need for bridging between a low-speed and high-speed bus. If the design includes a large number of low-speed peripherals, they can be connected on a separate Wishbone bus. The address width is 64 bits and the data width is configurable from 8 to 64 bits. Single cycle transfers are supported, provided the core supplying the data can respond fast enough. Like the PLB and OPB buses, arbitration is required if the multiple masters are present on the Wishbone; this must be implemented by the user.

The main advantages of the Wishbone interface are its simplicity and the fact that it is in the public domain. This encourages development and leaves users free to modify and improve the specification as they see fit. While the CoreConnect interface

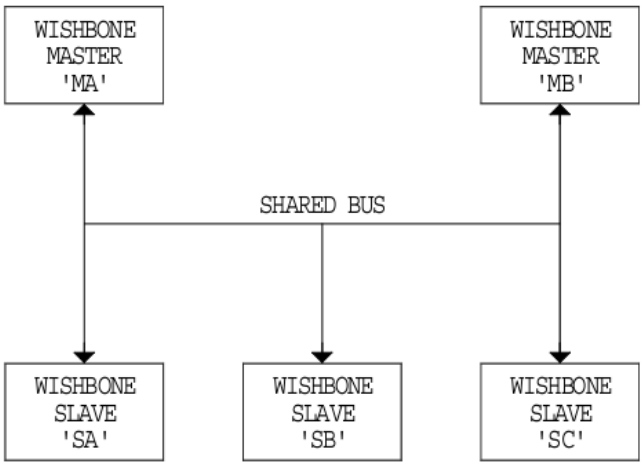


Figure 3.9 Wishbone bus architecture, reproduced from [36].

is comprehensive and free to use, development of the standard is ultimately controlled by IBM.

3.4 PERIPHERALS

A peripheral is essentially any module in the design that is connected to the processor. In a traditional embedded system these would typically be components that are connected externally to the microprocessor. One of the main aims of SoC design is to move as many of these components as possible inside the FPGA to allow greater flexibility. Three options are available to the the SoC designer: pay to use proprietary Intellectual Property (IP) cores, use free open source IP cores, or build the cores themselves.

3.4.1 Proprietary IP

A vast array of proprietary IP cores are available for use in SoC designs; many companies exist that generate all of their revenue through the production and sale of IP cores. However, due the large amount of effort involved in producing a core and the relatively small market size, the pricing often prohibits their use in academic work. A number of IP cores developed by Xilinx specifically for use on their own devices are supplied with the EDK tool, including:

- Timers – watchdog and general purpose
- Interrupt controller
- Memory controllers – BRAM, SDRAM, DDR RAM, Flash

- Communications modules – UART Lite, SPI, I^2C (evaluation), Ethernet 10/100 MAC (evaluation) and UART 16450/550 (evaluation)

The evaluation cores require the designer to pay a fee to use them in a design, since they include a large counter that will disable the core after 4 to 8 hours use, depending on clock speed. This can of course be avoided by resetting the device before the cores time out, but in many applications this may not be an option.

3.4.2 Open source IP

In academic work, using proprietary IP other than that included in EDK is impractical, both because of the expense and because it reduces the repeatability of the results by requiring others to purchase the core. Open source solutions are therefore often used when complex cores are required; OpenCores is the main repository of open source IP. A wide range of cores are available from OpenCores, including:

- Microcontrollers as discussed in section 3.2.4
- Communications cores – Ethernet 10/100/1000 MAC, I^2C, UART 16550, USB 1.1/2.0, CAN, IrDA, SPI and others
- Arithmetic and DSP cores – FFT, CORDIC, Floating point unit, multipliers and dividers, FIR blocks
- Encryption cores
- Wishbone controller cores

The drawback to using open source cores is that frequently they have only been verified on one or two devices or synthesis tools, so the user may need to spend some time modifying them to operate on their system. This in itself may prove difficult, as the majority of the cores are written in Verilog. Additionally, many cores either have no standard interface or use the wishbone interface, in which case they will need to be wrapped if used in an EDK design.

3.4.3 Custom IP

The final option for the designer is to build IP cores themselves. This is a somewhat time consuming process that is usually only worth pursuing when developing custom hardware coprocessors for use on the FSL.

The EDK package simplifies the process of creating custom IP with a tool for creating and importing peripherals. This allows the user to choose between an OPB, PLB or FSL peripheral and specify details about the interface, such as the width and

number of the inputs and outputs. The tool then generates code and an example application depending on the chosen interface, as well as driver files for the peripheral. This abstracts most of the detail of bus transactions from the user, speeding the design process significantly.

In the case of many communication protocols, writing a core completely is too involved, so an open source solution should be used. Since most open source cores have no standard interface or use the wishbone interface, the user may wish to wrap the peripheral for use on the OPB. This can be achieved with the OPB IPIF core, which is instantiated automatically by the peripheral tool.

The OPB IPIF core aims to simplify the process of designing custom peripherals for the OPB by providing a predefined interface between the user's logic and the bus architecture. The user defines how the IPIF interfaces to the user logic by including FIFOs, software accessible registers and interrupt controllers. The IPIF includes support for Direct Memory Access (DMA), by allowing the peripheral to act as a master on the OPB. By making use of the IPIF core, an open source core can be wrapped for the OPB by simply connecting its ports to the input and output registers or FIFOs of the IPIF core.

3.5 DESIGN TOOLS

Due to the complexity inherent in SoC designs, projects can be difficult to manage in a traditional design environment such as Xilinx's ISE. To this end, Xilinx have created the Embedded Development Kit (EDK) specifically for use in the design of SoC systems.

EDK is comprised of the Xilinx Platform Studio (XPS) and Xilinx Software Development Kit (SDK). The basic purpose of XPS is to provide a GUI for the assembly of the project, showing all of the components in the system and the buses they are connected to, and to integrate the software development process with the hardware development process. XPS provides mechanisms for easily managing the software running on the processor, with utilities for generating linker scripts and merging the compiled C code with the synthesised hardware platform. The GNU tool `make` is used to compile the software platform, simplifying the compile process. Information entered in the XPS GUI is stored in a number of configuration files specifying the design, and the experienced user may find it more productive to alter these files directly than use the GUI.

SDK is a software development platform based on the Eclipse IDE for Java development, useful for managing large software projects. The program also includes tools for JTAG debugging and profiling of embedded code.

While everything possible in EDK can be achieved in ISE, the framework offered by EDK speeds up the process greatly. Fully open source SoC designs must be developed

in a package such as ISE, and as such cannot benefit from the productivity advantages offered by EDK.

3.6 COMPARISONS

The details of SoC design have so far been presented without consideration of its advantages and disadvantages. In this section, open source and proprietary methods of designing SoC applications are compared, and the SoC design approach as a whole is compared with traditional embedded DSP design.

3.6.1 Open source vs proprietary designs

There are many advantages and disadvantages to each design approach, as touched upon in the preceding sections. While a design comprised solely of proprietary cores and components may be simpler and quicker to produce, the cost associated with using the IP is high. For this reason, such a design is only really viable in a commercial application with a short time-to-market requirement.

In an academic setting it is desirable to maximise the repeatability of the results and the reuseability of the code, and a completely open source design fulfils these requirements. However, the lack of specific SoC design tools and difficulty interfacing different modules may lead to a longer development time.

The best design approach is a compromise between the two methods, using a low-cost design environment and microprocessor such as EDK and the MicroBlaze to form the backbone of the system. Open source peripherals can then be instantiated in the design after being wrapped for the standard MicroBlaze interfaces. This approach combines the ease of design afforded by EDK with the freedom of open source IP and is highly repeatable, as most academic institutions will have access to the EDK software.

3.6.2 SoC vs traditional DSP

A SoC design has many advantages over a traditional embedded system, the most significant being the flexibility afforded to the designer. Moving more of the system inside the FPGA allows the designer to make major changes to the hardware architecture right up to the end of the design process. In a traditional embedded design, most of the hardware infrastructure is unable to be modified once the PCB layout has been finalised. If the software design requires more RAM or a processor with a floating point unit, changing the hardware platform becomes very time consuming and costly.

Of course, traditional embedded systems also have their advantages, principally their low cost. The large FPGAs required for complex SoC systems are significantly more expensive than the individual components that would make up a traditional

embedded system. Low cost FPGAs such as the Xilinx Spartan series can compare favourably to traditional systems but lack the processing power required for computationally intensive DSP algorithms, instead being more suited to simpler applications.

Chapter 4

IMPLEMENTATION

The system consists of six major blocks: the MicroBlaze processor, bathymetry co-processor, USB 1.1 and RS232 communications peripherals, and internal and external RAM, as shown in Figure 4.1. The MicroBlaze is the core of the system, and communicates with the SDRAM, USB and RS232 peripherals over the OPB. Most of the details of bus transactions are hidden from the user, with functions provided to read and write to locations within the peripherals. Information is transferred to and from the coprocessor via the FSL, a FIFO-based interface.

Code to be executed on the processor is compiled and merged with the FPGA programming file to set the initialisation strings for the internal Block RAM (BRAM). The MicroBlaze then fetches instructions and data from the BRAM over the Local Memory Bus (LMB), which allows single cycle memory access. For larger programs, sections can be specified to reside in external memory in the linker script, and are downloaded via the debugger interface once the FPGA has been programmed.

All peripherals on the the OPB are memory mapped, meaning that reading or writing to a peripheral is functionally equivalent to reading or writing to a memory element from the user's point of view.

4.1 USB

One of the essential requirements of the design is a means to transfer data between the device and the host PC. This was achieved using the Universal Serial Bus (USB) [37]. An overview of the USB protocol is presented here, [38] is a good resource for a more in-depth explanation.

Three data rates are available in USB: 480 Mbps (high-speed), 12 Mbps (full-speed) and 1.5 Mbps (low-speed). USB 2.0 supports all three, while USB 1.1 only supports the two lower speeds. USB is completely host controlled, meaning that transfers can only occur when requested by the host. Data transfers centre around buffers in the USB device called endpoints. A USB device will always have a control endpoint, which

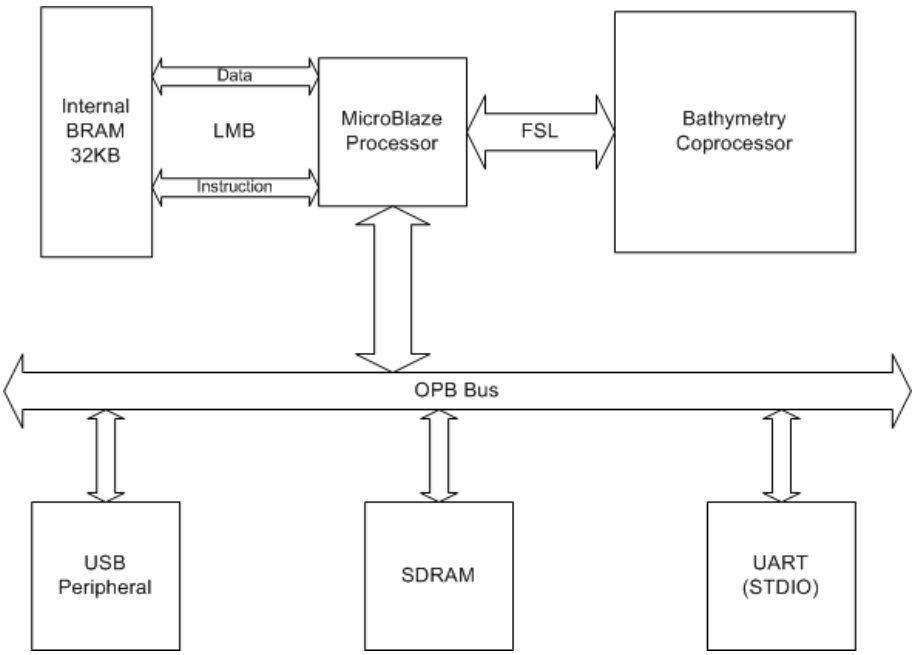


Figure 4.1 Block diagram of MicroBlaze system.

is bi-directional and responsible for enumerating with the host and initiating of data transfers. Data endpoints can have one of three types:

- Interrupt endpoints are used for small, non-periodic data transfers. When an interrupt is generated it is queued until the interrupt endpoint is polled.
- Isochronous endpoints are typically used for time-sensitive information, as they occur periodically and guarantee access to bus bandwidth.
- Bulk endpoints are used for sending large amounts of data. Bandwidth and latency are not guaranteed as bulk transfers are allocated after all other transactions on the bus. Bulk transfers are only supported at high and full speeds.

Data endpoints are always unidirectional and named with reference to the host; IN endpoints send data to the host, while OUT endpoints receive data from the host.

4.1.1 Components of a USB system

On the device side of a USB interface, the system design differs between USB 1.1 and USB 2.0. USB 1.1 is comprised of three main parts: the transceiver, a PHY, and the function core itself, while a USB 2.0 device usually consists of an external PHY that communicates with the function core via the USB 2.0 Transceiver Macrocell Interface (UTMI).

4.1.1.1 USB 1.1

The most basic element in the USB 1.1 device is the transceiver, which simply interfaces between the differential serial signals sent over the cable and the digital USB interface to the PHY. The transceiver is the only component in the USB 1.1 system that is external to the FPGA.

The PHY converts between the serial USB interface and the parallel connection to the function core, a simplified version of the UTMI. This involves performing bit stuffing and unstuffing, and NRZI encoding and decoding[37].

The most complex part of the USB system is the function core. It performs enumeration with the host fully implemented in hardware, eliminating the need for an external microcontroller. Endpoints are implemented as FIFOs external to the function core.

4.1.1.2 USB 2.0

USB 2.0 provides a high-speed transfer rate of 480 Mbps on top of the full-speed (12 Mbps) and low-speed (1.5 Mbps) data rates supported by USB 1.1. As this requires a much higher clock speed than can easily be synthesised in an FPGA the PHY is implemented on an external chip.

The external PHY performs the same function as both the transceiver and the PHY in the USB 1.1 system, converting the differential USB signal to a parallel UTM interface. The function core can run at a lower clock speed inside the FPGA, and communicates with the PHY over the UTMI. The function core is similar to the USB 1.1 function core, but includes extra logic to deal with high-speed transfers and uses shared memory rather than FIFOs for the endpoints.

4.1.2 OpenCores USB 1.1 function core

The need for an external PHY for USB 2.0 transfers means that a custom expansion board would have to be made implement it, so for the sake of simplicity a USB 1.1 system was used. Xilinx does not provide a proprietary USB core with the EDK package, due to the large device area needed to implement the protocol on chip. A free USB 1.1 core and PHY with functionality as described above is available from OpenCores, and provides the basis of the USB interface in the design. A block diagram of the USB 1.1 system is shown in Figure 4.2 below.

4.1.2.1 Core configuration

The key components of the core are the endpoint FIFOs, the PHY, and the function core itself. The PHY is internal to the top level of the core, so details of the UTMI are

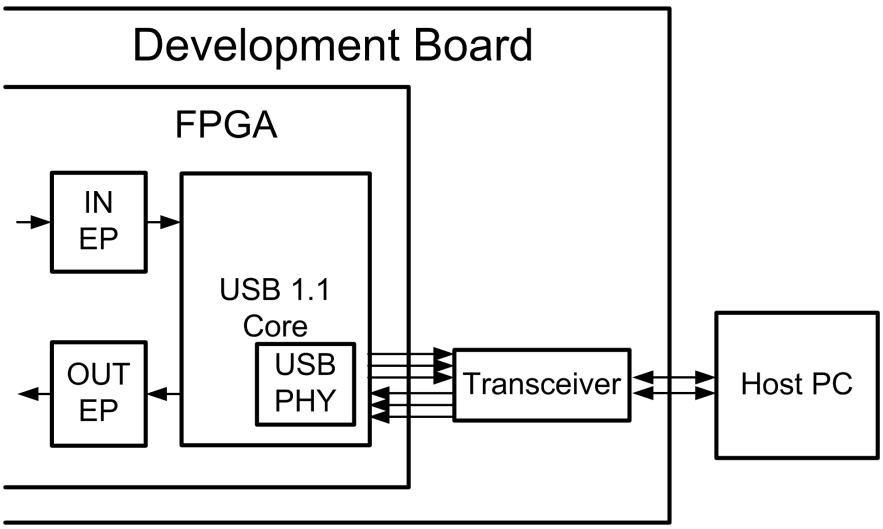


Figure 4.2 Block diagram of USB 1.1 system.

abstracted from the user. The top level design is therefore relatively simple, consisting of an external interface to the transceiver and internal interfaces to the endpoint FIFOs.

The core can support up to 8 endpoints, the first of which is always a bi-directional control endpoint, responsible for enumeration with the host and initiation of transfers. The remaining endpoints can be configured as either in, out or bi-directional and with one of four endpoint types: bulk, interrupt, isochronous, or control. Only control endpoints are permitted to be bi-directional. The endpoint interface itself consists of a configuration input, data inputs and outputs and FIFO full and empty inputs, depending on the direction of the endpoint.

In this application, the core was configured with the compulsory control endpoint, one bulk IN endpoint and one bulk OUT endpoint. FIFOs were used for the bulk endpoints — 2 kB for IN and 4 kB for OUT, as more data is received than sent back to the host. The core includes a ROM that holds descriptors for the device and endpoints, these are sent to the host when it interrogates the device. Device descriptors hold information about the vendor and product IDs, the number of endpoints and the speed of the device. Each endpoint then has its own descriptor detailing the address, type, direction and maximum packet size for the endpoint. These must be configured to match the endpoints.

4.1.3 Race condition

The core was synthesised and tested on the development board, but unfortunately the design appears to have a race condition that causes enumeration to fail occasionally. The effects of the race condition can be made to disappear by tweaking mapping and place and route parameters, but it will often reappear when the design is changed, even

if the change does not involve any of the USB core components.

This proves to be rather frustrating, but tracking down the root cause of the race condition could be quite a laborious process, especially as the core is written in Verilog with limited documentation. It may be that the design simply requires extra constraints to be added to ensure consistent results from the mapping and place and route stages. This issue is discussed further in Chapter 5.

4.1.4 Wrapping OpenCores USB 1.1 for OPB

In order to use the OpenCores USB core with the MicroBlaze it must be wrapped for use as a peripheral on the OPB. This was achieved through the use of the OPB IPIF core, described in section 3.4.3.

The FIFO functionality of the IPIF core would be ideal in this application, but it only supports a 32-bit data width — much larger than the 8 bits required by the USB — and does not currently include support for independent clocks. The USB core has a fixed clock speed of 48 MHz, but ideally the system clock would run at 100 MHz to achieve faster execution of the processing algorithms.

A dual clock design was implemented, using independent clock FIFOs to interface between the two clock domains [39]. The IPIF interface provided software access to read and write registers, the contents of which were then transferred to the FIFOs using an FSM. Unfortunately the design would not meet timing so the USB peripheral was scaled back to a single clock design. Under normal circumstances an FPGA design should be easily synthesisable with more than one clock domain, so it is suprising that the design would not meet timing, but it is possible that there are insufficient resources to route the two separate clock signals. The OPB IPIF guide [40] advises that clock enables should be used instead of different clock domains, so perhaps there is some known shortcoming in the IPIF design that prevents dual-clock designs from synthesising.

A single clock design uses the OPB clock to provide timing for the USB core. As the OPB clock always runs at the same frequency as the MicroBlaze, the whole system runs at 48 MHz. In some systems with DDR memory this would present a problem as DDR typically has a minimum operating frequency — 66 MHz in the case of Micron MT46V16M16TG-75 used on a Virtex-II^{TM} board that was evaluated — however, the Virtex-II Pro^{TM} board used in the project uses slower SDRAM, which is not subject to a minimum clock frequency constraint.

The interface to the MicroBlaze processor is simply four software accessible registers: read, write, and two registers holding information about the data level in each endpoint FIFO. The data level registers are updated every clock cycle with the output from the endpoint FIFOs; updating the read and write registers is slightly more complex. Each register is controlled by a small FSM and a status flag. Since the status flag

is altered in two separate processes that run in parallel, simply using a signal as a flag will produce multi-source errors during synthesis as both processes could potentially try to drive the signal at the same time. To avoid this, flags are implemented using the simple design in Figure 4.3 below.

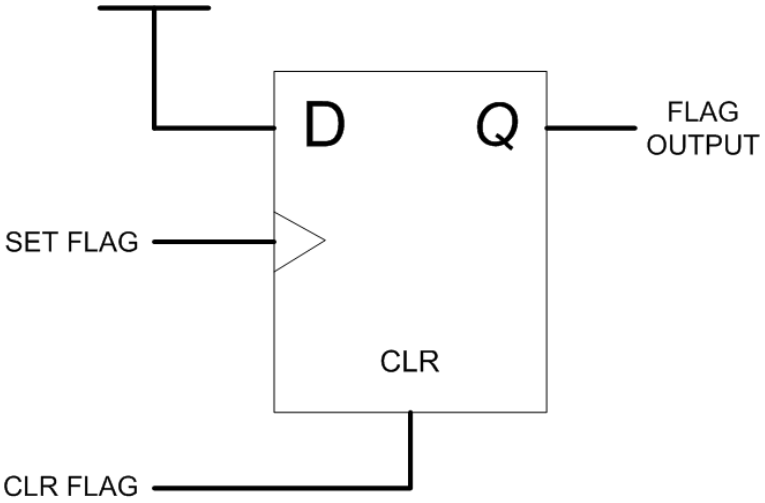
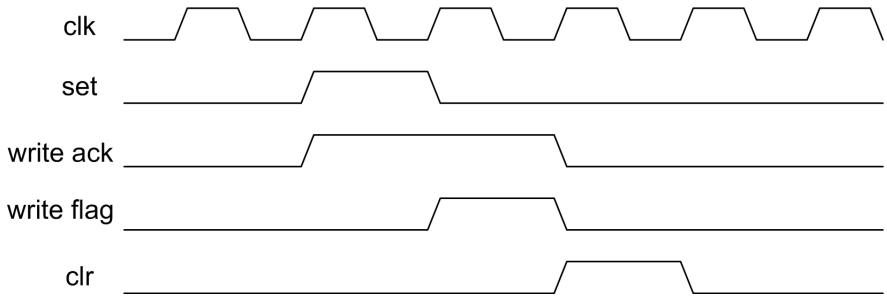


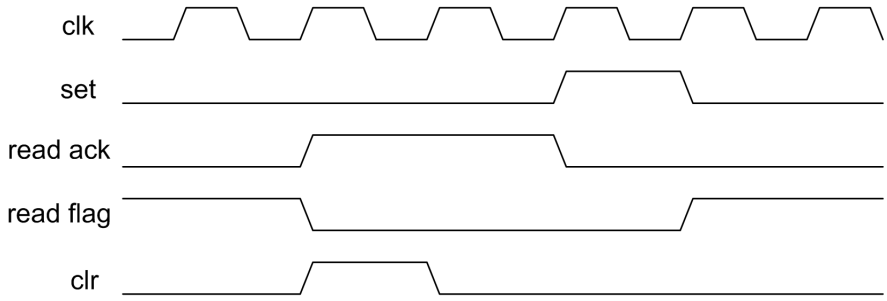
Figure 4.3 Circuit diagram for flag design.

In the case of the write flag, the set signal is driven by the process writing into the write register. The write FSM then recognises that the flag has gone high, copies the data from the write register into the write FIFO and asserts the clear signal. Likewise, when data is read from the read register by the MicroBlaze, the read flag is cleared. The read FSM then recognises that the flag has gone low, copies an entry from the FIFO into the read register and asserts the set signal. This process can be seen in Figure 4.4 and a block diagram of the entire USB peripheral is shown in Figure 4.5 below.

Since both the set and clear inputs have the ability to alter the output and are controlled by separate processes there is potential for a race condition to occur. This is avoided by careful programming of the FSMs that control the flags. For example, during a write operation the flag is set when data is written to the register. The FSM that clears the flag requires the ACK signal to be LO before it asserts the CLR signal while the FSM that sets the flag can only do so while the ACK signal is HI. The ACK signal itself is asserted in response to an OPB read or write command, and remains HI for the duration of the transfer. This setup ensures that a race condition cannot occur, since the flag can only be cleared once the write transfer has been completed.



(a) Write operation.



(b) Read operation.

Figure 4.4 Flag signals during read and write operations.

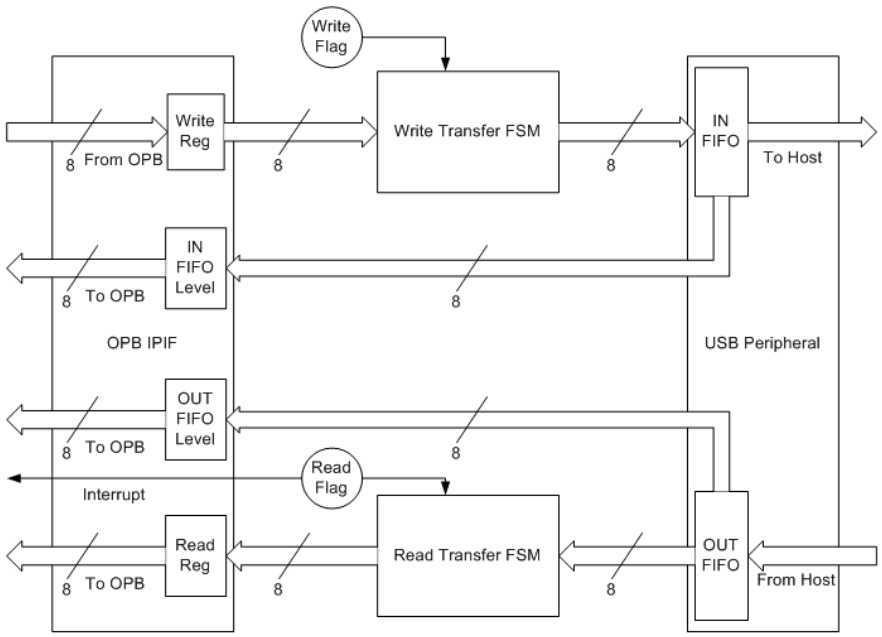


Figure 4.5 Block diagram of USB peripheral.

4.2 MICROBLAZE

The MicroBlaze processor is a complex system component that provides much of the functionality in the design. It is responsible for reading data packets from the USB core and responding to the host PC appropriately, as well as buffering data, interfacing to the coprocessor, and performing some basic computations on the correlator output. In this section the general setup of the MicroBlaze is presented and the tasks it performs are detailed.

4.2.1 Processor configuration

One of the advantages of the MicroBlaze processor is its flexibility. Depending on the requirements of the application, hardware accelerated modules can be added to the basic processor to speed up the execution of more complex operations. Once the modules are added, extra arguments are passed to the compiler in the command line, telling it to use the hardware accelerated instructions over the library routines. Of course, these extra modules add to the device area required by the processor.

In this application, the MicroBlaze is configured with a hardware multiplier and divider, offering a significant increase in speed over equivalent software library routines. Since the software platform uses a large number of bit shifting operations, both for division by factors of two and for converting numbers between 8-bit and 32-bit formats, a barrel shifter was included to allow this operation to be carried out in one instruction.

Code for the MicroBlaze processor was written in C for ease of implementation. Since most of the computationally intensive processing is done directly in hardware it is not essential that the MicroBlaze code runs as fast as possible so it was not written in assembler. Additionally, should the project be migrated to a PowerPC processor in the future, the code will not have to be completely re-written.

4.2.2 Program operation

In this application, the basic requirement of the MicroBlaze processor is to shuttle data between the USB core and the coprocessor doing the sliding correlation. Since the coprocessor takes approximately 11500 cycles to process a correlation window of 32 samples, there is plenty of time for the MicroBlaze to perform other tasks in the interim. The basic program operation is as follows:

1. Write one sample from each ping to the correlator.
2. Using previous correlator output, calculate the coarse delay by fitting quadratic.
3. Interpolate values of real and imaginary signals using calculated delay.
4. Normalise the correlation result.

5. Write results into output buffer.
6. Perform a USB transfer.
7. Prepare correlator input for next iteration.
8. Update the integrate-and-dump sum with new inputs values.
9. Read output of correlator.
10. Return to step 1.

4.2.2.1 Ping-pong buffering

In order to keep a constant flow of data to the correlator a ping-pong buffering scheme was adopted. The program transfers data over USB into one set of buffers while the coprocessor operates on data in an identical set of buffers. This allows raw and processed data to be transferred to and from the host without breaking the flow of data to the coprocessor. The total memory used for buffering is 8 kB — each 256-sample sonar ping is 1 kB, two pings are stored for each correlation, and they must be buffered in and out. Traditionally, this type of buffering would be handled by DMA. It is possible to implement DMA in the OPB IPIF core, but this only facilitates memory transfers to memory on the OPB, i.e., external memory. Since the data is small enough to be stored in fast-access internal memory, this microcontroller arbitrated pseudo-DMA scheme is implemented.

4.2.2.2 Quadratic fitting

Finding the peak in magnitude of the correlation result gives the delay between the two windows being correlated. Conventionally, the magnitude would be calculated as shown in (4.1), but the computationally intensive square root operation can be eliminated as only the location of the peak value is of interest.

$$\text{Mag} = \sqrt{\text{Re}^2 + \text{Im}^2} \tag{4.1}$$

The probability that the correlation peak resides exactly at a specific sample is slim, so it is necessary to fit a curve to the points about the peak and use this to find a more accurate estimate of the peak location. A simple interpolation scheme is to fit a quadratic to the three points about the peak — the maximum and the samples either side of it.

It was decided to use the MicroBlaze processor to fit the quadratic and find the peak, as it is an operation that can easily be executed while waiting for the correlator output. The quadratic function takes as an input the three squared magnitude values

described above. This introduces an interesting design decision, since the magnitude can either be recalculated from the correlator output or the correlator modified to output the magnitude as well as the real and imaginary values. Since a MicroBlaze hardware multiply operation takes 3 clock cycles and an FSL read operation takes 2, it is more efficient to output the required values from the correlator block, where they have already been calculated.

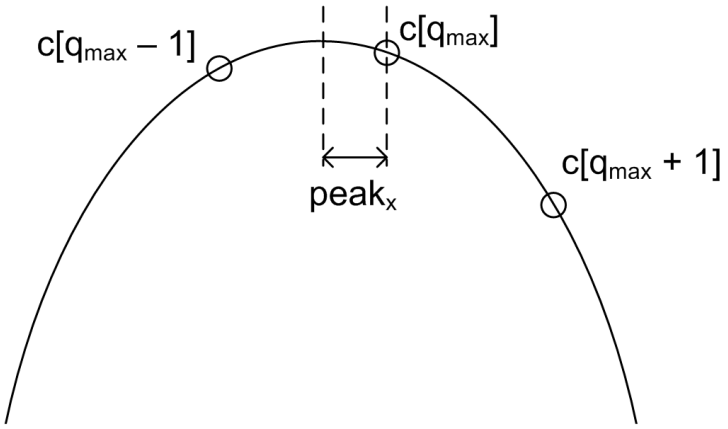


Figure 4.6 Quadratic curve fitted to three samples about maximum.

Consider the curve in Figure 4.6, where $c[q_{\max}]$ is the sample returned by the coprocessor as the maximum for a correlation window. The location of the peak relative to the maximum sample is calculated as

$$\text{peak}_x = \frac{-0.5b}{a} \tag{4.2}$$

where

$$a = 0.5\text{Mag}(c[q_{\max}+1]) + 0.5\text{Mag}(c[q_{\max}-1]) - \text{Mag}(c[q_{\max}]) \tag{4.3}$$

$$b = 0.5\text{Mag}(c[q_{\max}+1]) - 0.5\text{Mag}(c[q_{\max}-1]) \tag{4.4}$$

Substituting into (4.2) and simplifying gives

$$\text{peak}_x = \frac{1}{2}\frac{\text{Mag}(c[q_{\max}-1]) - \text{Mag}(c[q_{\max}+1])}{\text{Mag}(c[q_{\max}-1]) + \text{Mag}(c[q_{\max}+1]) - 2\text{Mag}(c[q_{\max}])} \tag{4.5}$$

Implementing (4.5) in a fixed point system requires some minor modifications. Firstly, the square root operation in the magnitude equation will not affect the result, so it is removed to speed execution. By the nature of the quadratic fitting process, (4.5) will always give a result between -0.5 and 0.5. If (4.5) is implemented exactly

as shown above the result will always be zero, due to integer rounding. It is therefore necessary to either scale up the numerator or scale down the denominator by the level of precision required, prior to computing (4.5). Since scaling up the numerator will cause overflow in the majority of cases, the denominator is scaled down by 32768 so that the result will be a Q-15 scaled integer [10]. Because the result will always be between -0.5 and 0.5 the factor of 2 in the denominator of (4.5) can be removed to fully utilise the dynamic range of the integer. Equation (4.5) is now expressed as:

$$\text{peak}_x = \frac{1}{32768} \frac{(\text{Mag}(c[q_{\max} - 1]))^2 - (\text{Mag}(c[q_{\max} + 1]))^2}{(\text{Mag}(c[q_{\max} - 1]))^2 + (\text{Mag}(c[q_{\max} + 1]))^2 - 2(\text{Mag}(c[q_{\max}]))^2} \quad (4.6)$$

The resulting 16-bit signed integer is scaled between -0.5 (-32768) and 0.5(32767). This scaling must be taken into account in any further operations on the result.

4.2.2.3 Interpolation

Once the delay between the two pings has been found, the real and imaginary signals must be interpolated to find an estimate of the values for that specific delay as shown in Figure 4.7. With the output available from the correlator, three simple interpolation schemes are possible: nearest neighbour, linear and quadratic. Due to the long range of the sonar, small errors in interpolation are amplified when generating the height estimate, so it is desirable to keep the interpolation as accurate as possible. The nearest neighbour approach is therefore unacceptable due to the large error involved. The implications of the remaining two approaches will be discussed in Chapter 5; for now the implementation of each scheme is presented.

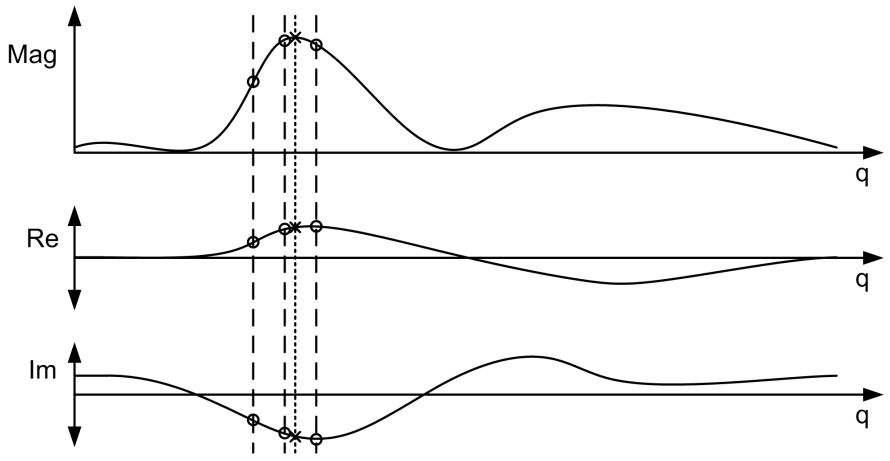


Figure 4.7 Once the magnitude-of-correlation peak is found, the real and imaginary components of the correlation result are interpolated.

Linear interpolation is the simpler of the two methods. The point to be interpolated is known from the quadratic-fitting process to lie between -0.5 (-32768) and 0.5 (32767) samples relative to the maximum returned by the correlator. The difference between

the two points to be interpolated is found and halved to correspond with the scale of the delay estimate. This result is then simply multiplied by the delay estimate and added to the middle sample.

$$y[q_{\text{int}}] = y[q_{\max}] + \text{peak}_x \left(\frac{y[q_{\max}+1] - y[q_{\max}]}{2} \right) \tag{4.7}$$

Quadratic interpolation is slightly more involved. A quadratic function is fitted to the three points in a similar manner to section 4.2.2.2. The curve is defined by the standard quadratic function:

$$y = ax^2 + bx + c, \tag{4.8}$$

where

$$a = 0.5c(\max+1) + 0.5c(\max-1) - c(\max) \tag{4.9}$$

$$b = 0.5c(\max+1) - 0.5c(\max-1) \tag{4.10}$$

$$c = c(\max) \tag{4.11}$$

The equation is then evaluated with the result from the quadratic fitting function for x, which must first be divided by two so that it ranges from -0.5 to 0.5.

Since ultimately the phase will be calculated to generate a fine estimate for the delay, another common approach is to calculate the phase at the two relevant points and use linear interpolation. Because the phase should vary linearly across the correlation peak, this method provides a good estimate of the phase at the point of maximum magnitude [41]. However, since the magnitude information used to generate the coherence map is not contained in the phase, the magnitude must be interpolated separately. To reduce complexity only magnitude interpolation was used; the phase is then calculated from the interpolated results.

4.2.2.4 Normalisation

The correlator will return a result irrespective of whether the two inputs are in any way correlated, so to construct an accurate height map it is useful to have a measure of how meaningful each result is. This is achieved by normalising the result as discussed in section 2.2. The normalised correlation equation is reproduced here.

$$\rho_{d_1 d_2}(q) = \frac{R_{d_1 d_2}(q)}{\sqrt{R_{d_1 d_1}(0) R_{d_2 d_2}(0)}} \tag{4.12}$$

where,

$$R_{d_1 d_1}(0) = \sum_{m=0}^{M-1} x_1(m)^2 + y_1(m)^2 \tag{4.13}$$

$$R_{d_2 d_2}(0) = \sum_{m=0}^{M-1} x_2(m)^2 + y_2(m)^2 \tag{4.14}$$

As the window is only moving by one sample with each iteration, the autocorrelation can be produced efficiently by employing an integrate-and-dump approach. If a buffer of the elements in the sums above is stored, the autocorrelation can be simply evaluated when the window is moved by subtracting the oldest value in the buffer and adding the new value. This method is employed in [14].

On the MicroBlaze a circular buffer is set up to store the previous elements with a pointer to the oldest. When the window is moved the oldest value is subtracted from the running sum then overwritten with the new value, which is added to the sum. The pointer is then moved to the next value in the buffer, which then becomes the oldest.

Evaluating (4.12) requires a square root operation to be performed on the denominator, which is in 16-bit fixed-point format. The C library routine `sqrt()` uses floating point arithmetic to find the square root. Converting to floating-point results in inefficient code as the MicroBlaze lacks an FPU in its current configuration, so a fixed point square root routine was implemented. The routine proposed by Turkowski [42] uses a method similar to long-division to find the root of a 32-bit fixed point number with 30 fractional bits. Since the product of the Q-15 multiplication is Q-30, the Turkowski method can be used without modification before the product is converted back to Q-15. For more information on the algorithm itself the reader should consult [42] and [43].

4.2.2.5 USB transfers

Once during every correlation loop, the `USB_Transfer` function is called, which initiates the transfer of a data packet if data is available. The operation of the function is as follows:

1. Read from `OPB_USB` to check number of entries in receive FIFO. Exit if less than 3.

2. Read 3 bytes from `OPB_USB`. This is the packet header. The first byte is always 0xAB and indicates a start of packet. The next byte defines the transaction type, and can have 5 possible values:

   - 0x01 — Write to Ping 1
   - 0x02 — Write to Ping 2

- 0x10 — Read from Ping 1
- 0x20 — Read from Ping 2
- 0xFF — Status Request

The final byte specifies the index of the array to start read or writing at, unless the packet header is a status request, in which case it is ignored.

3. Depending on the transaction type, 32 bytes starting at the specified index are written to or read from the USB FIFOs. If the transaction is a status request, 3 bytes are sent back: The number of values written to Ping 1, the number of values written to Ping 2, and the number of entries in the receive FIFO.

The size of the data packets was set to 35 bytes to limit the amount of time that is spent in the `USB_Transfer` function, as ideally it must finish executing before the correlation result is available.

4.3 CORRELATOR

As discussed in Chapter 2, the core of the interferometry process is a sliding correlator followed by a peak estimator. Pings of two images from separate receiver rows are correlated as shown in Figure 4.8 below, then the peak of the correlation is found to give an estimate of the delay between the two pings. The correlation equations derived in section 2.2 are reproduced here:

$$c(q) = \sum_{m=0}^{M-1} d_1(m) d_2^*(m+q) \tag{4.15}$$

where

$$d_1(m) = x_1(m) + jy_1(m) \tag{4.16}$$

$$d_2(m) = x_2(m) + jy_2(m) \tag{4.17}$$

substituting into 4.15 gives

$$c(q) = \sum_{m=0}^{M-1} \left[x_1(m) + jy_1(m)\right] \left[x_2(m+q) jy_2(m+q)\right] \tag{4.18}$$

$$= \sum_{m=0}^{M-1} x_1(m)x_2(m+q) + y_1(m)y_2(m+q) + j \sum_{m=0}^{M-1} y_1(m)x_2(m+q) - x_1(m)y_2(m+q), \tag{4.19}$$

where M is the size of the correlation window; in this case 32 samples. Performing the correlation for values of q between 0 and 15 results in an array of 16 correlation values. Finding the position of peak value in the array gives the delay between the pings at the first sample in the window. The window is moved along the ping one sample at a time, as shown in Figure 4.8 above, building up an array of delays that describe the height of the sea-floor.

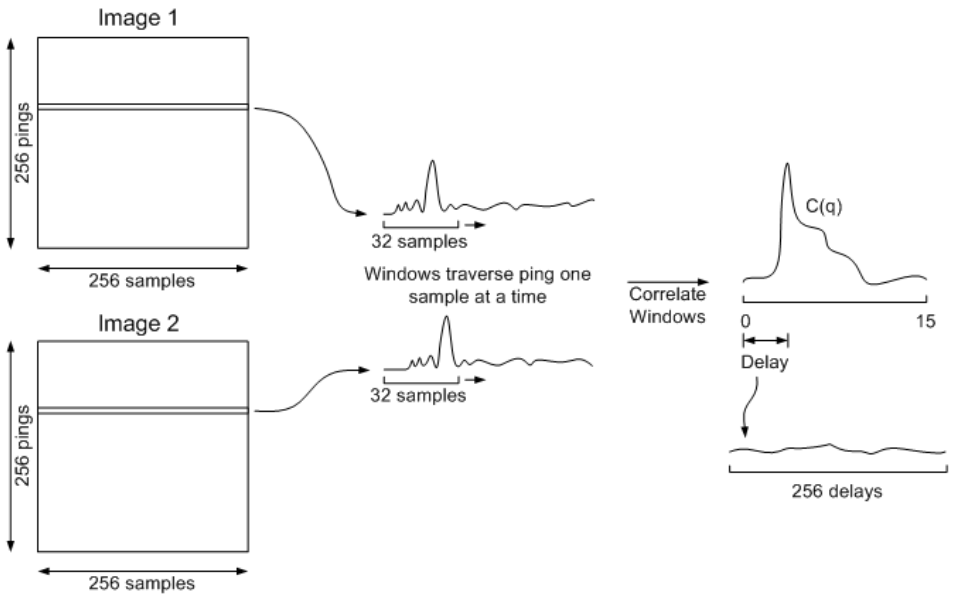


Figure 4.8 Correlation of pings from two separate receiver rows.

4.3.1 FSL Interface

The correlator coprocessor communicates with the MicroBlaze processor via the Fast Simplex Link (FSL), discussed in detail in Chapter 3. The FSL is a uni-directional link, so for two-way communications between the coprocessor and the MicroBlaze, two FSLs are required. Each FSL is essentially an asynchronous FIFO, supported by custom instructions that allow integration into the pipeline. As such, the send and receive framework implemented in the coprocessor is not complicated.

4.3.1.1 Read

The state machine reads two new values each time a correlation is computed. In a read operation, the state machine waits for the `FSL_Exists` signal to go high, indicating that data is available to be read from the FSL FIFO. It then asserts `FSL_Read` and writes the output value into BRAM.

4.3.1.2 Write

At the end of each correlation the state machine writes the results back to the MicroBlaze. To write data to the FSL, the state machine waits for the `FSL_Full` signal to go low, indicating that there is space in the FIFO. It then presents the data to be written on the data line and asserts the `FSL_Write` signal.

4.3.2 Data buffering

Since the 32-sample correlation window only moves by one sample with each iteration, it makes sense to only send the newest sample and keep the old samples in a circular buffer. The BRAM block used in the coprocessor has a 10-bit address bus, so creating a 32-element circular buffer is simply a matter of creating a 5-bit base address and a 5-bit data pointer for each circular buffer, concatenated to form a 10-bit address. At the beginning of each correlation the new sample is written over the oldest sample and the pointer is incremented as shown in Figure 4.9.

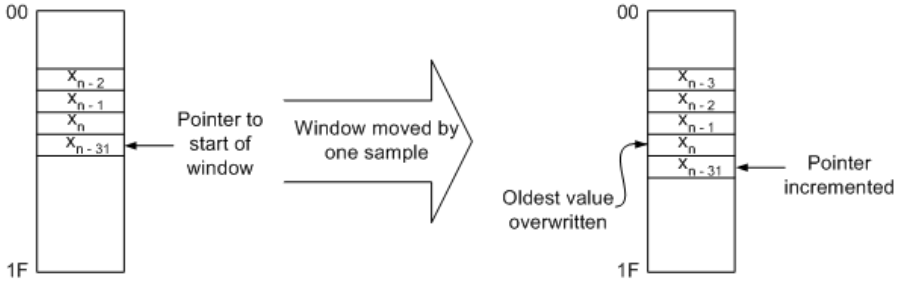


Figure 4.9 Using circular buffers for efficient windowing.

Allowing the pointer to overflow means that when m and q are added to the pointer it will automatically wrap from the end to the start of the buffer. Care must be taken to ensure that a memory access is not executed if $m + q > 32$ since this will have the effect of re-reading a sample. In these cases the operand can be simply set to zero as it is unlikely that the result will be important.

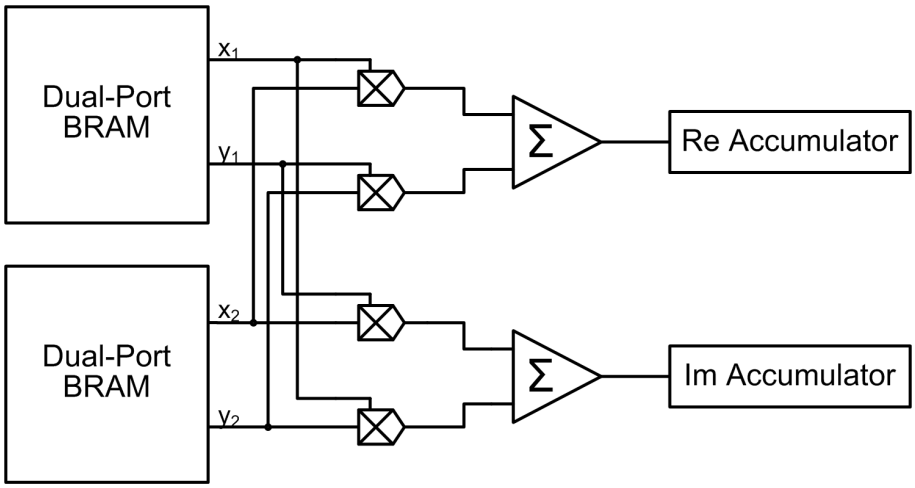
Four such buffers are set up to store the 16-bit real and imaginary values of the two receiver pings. The ping data arrives in pairs as 32-bit compound complex integers, where the most significant word is the real value and the least significant the imaginary. The integers are split into their real and imaginary parts and written into the dual port BRAM. The dual port architecture is useful as it allows both operands for the multiplication stage to be fetched in one clock cycle.

Besides correlation windowing, circular buffering has many other uses in digital signal processing in applications such as FIR filtering. The main advantage over other

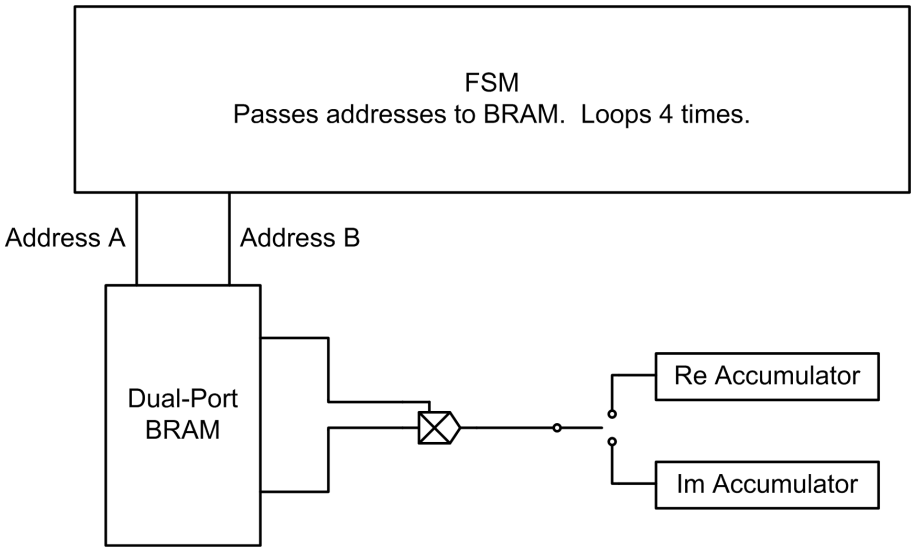
buffering schemes is the low computational overhead, as incrementing a pointer once is much more efficient than moving every element in a buffer.

4.3.3 Parallel vs serial

A correlator can easily be implemented in hardware; two possible architectures are shown in Figure 4.10 below.



(a) Parallel implementation.



(b) Serial implementation.

Figure 4.10 Comparison of serial and parallel implementations of the correlator MAC block.

The first implementation takes advantage of the parallel processing power of FPGAs by executing the multiplication and addition stages in parallel, but is more complex to implement and uses more resources. The serial implementation is smaller, using

only one block RAM and one 18x18 multiplier to loop through the multiplication and addition stages. This makes it significantly slower than the parallel implementation but much simpler to implement. A comparison of resource requirements is shown in Table 4.3.3 below.

Resource	**Parallel**	**Serial**
BRAM	2 (5%)	1 (2%)
MULT18x18	4 (9%)	1 (2%)
Logic Slices	unknown	532 (10%)
Execution time for one window	approx 2900 cycles	approx 11500 cycles

Table 4.1 Comparison of correlators with parallel and serial MAC blocks.

4.3.4 State machine

Due to development time constraints, the serial correlator was chosen for its simplicity, and implemented as a Finite State Machine (FSM). The simplified state diagram in figure 4.11 shows its operation.

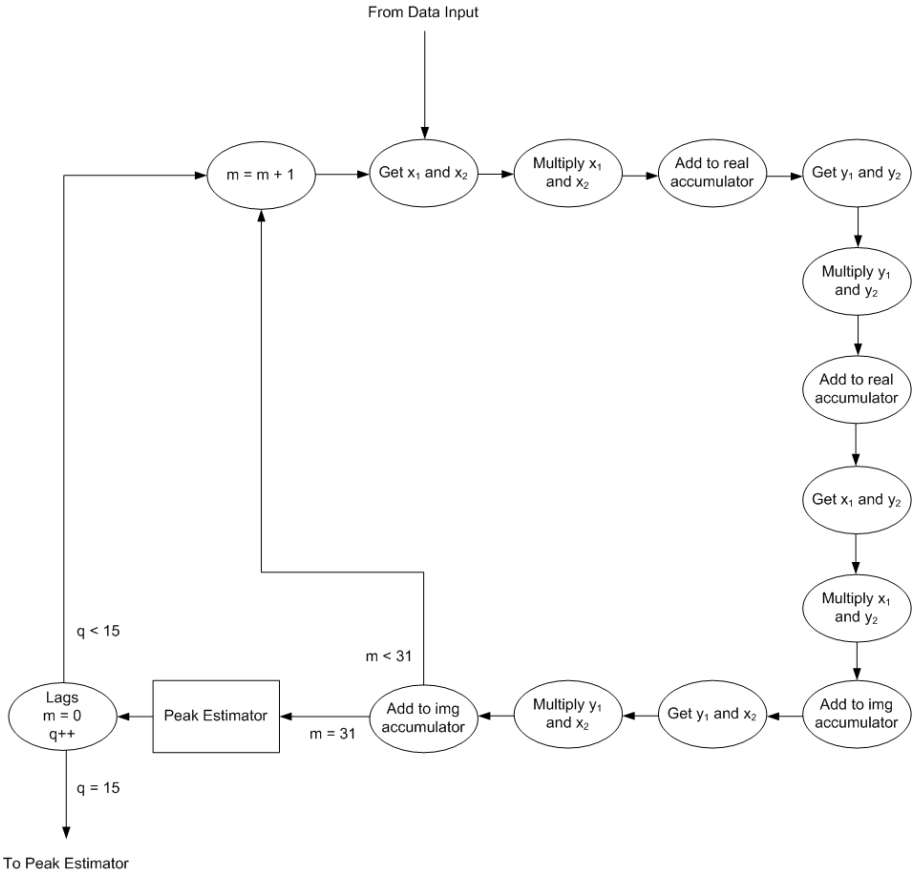


Figure 4.11 Finite State Machine showing correlator algorithm operation.

The state machine waits until two input values have been read into RAM from the FSL interface, then begins computing the correlation. Operands for the multiplier are

read from the BRAM and passes operands to the multiplier, then the multiplier result is added to accumulators for the real and imaginary results. The process is repeated 32 times to compute the real and imaginary sums for each of 16 lags.

4.3.4.1 Overflow handling

When computing a large number of multiply and accumulate (MAC) operations, the system must be designed to avoid overflow. By using a fixed point Q-15 number representation for 16-bit integers, overflow resulting from multiplication is eliminated, as multiplication operands are always less than 1. Overflow can still occur in addition operations however, so the accumulators in the system must incorporate extra "guard bits" to prevent addition overflows. Since each accumulator must hold the results of 64 accumulations, in the worst case – all ones – 6 extra bits will be needed, making each accumulator 38 bits long. Once the MAC operations are complete, the top 16 bits of the accumulator are passed to the next stage.

4.3.4.2 Peak detection

Equation (4.19) is evaluated for values of q from 0 to 15. After each iteration the result is passed to the peak detection algorithm. The square of the magnitude is calculated and saved in a temporary register. The magnitude is then tested against the previous maximum. If it is larger, it is saved as the new maximum, and the previous result is saved as the sample to the left of the maximum. A flag is set to tell the peak detection algorithm to save the next result as the sample to the right of the maximum on the next iteration. If the result is smaller than the previous maximum, it is still saved in case it is needed in the next iteration. The process is illustrated in figure 4.12 below.

Once (4.19) has been calculated for all 16 values of q the three points about the maximum of the magnitude curve, the same three points in the real and imaginary curves and the index of the maximum are passed back to the FSL interface.

4.3.5 Timing considerations

In the configuration described above, one correlation takes approximately 11500 clock cycles. At a clock speed of 48 MHz this means that processing a 256-sample ping will take 61 ms. Since the sonar sends pings at a rate just under 15 per second, each sample must be processed in under 66 ms. Assuming that ping data can be moved on and off the chip fast enough, this correlator configuration should provide a real-time solution.

Ideally, the system would process all of the image, increasing the ping length to 2048 or 4096 samples. To keep up with this increased data rate, the speed of the correlator would need to be increased dramatically. There are a number of ways this could be achieved, discussed in Chapter 5.

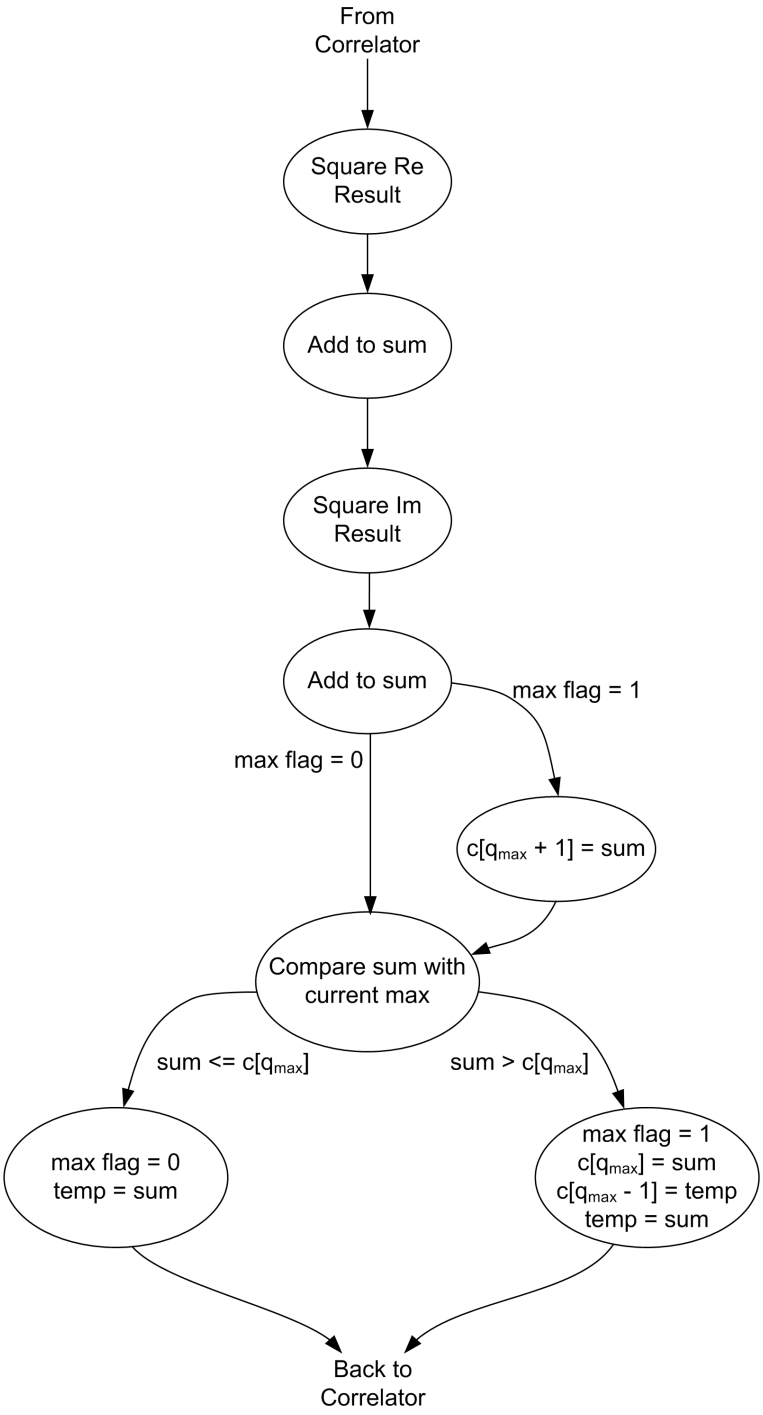


Figure 4.12 Finite State Machine showing peak detection algorithm operation.

4.4 HOST-SIDE DRIVER

The interface to the device from the host PC uses the libusb project to provide basic read and write functionality to the device. The host-side driver builds on the bulk read and bulk write functions by providing functionality to write or read an entire ping to or from the device. Ideally the driver program will read and write continually, making use of the ping-pong buffering scheme on the chip.

4.4.1 libusb

The libusb tool is an open source project for Linux aiming to develop an API to allow users to write their own USB device drivers. To this end basic functions are provided to scan the USB devices attached to the host, claim devices and interfaces, and transfer data to and from the device.

Ideally, isochronous data transfers would be used to ensure that the device keeps up with the constant flow of data from the sonar, but only bulk and interrupt transfers are supported by libusb. Since the amount of data being sent to and from the device is quite large, bulk transfers were used. The maximum packet size for a bulk transfer is 64 bytes, but a frame can be made up of many packets. The functions `usb_bulk_read()` and `usb_bulk_write()` abstract the detail of this from the user, breaking up large input data into appropriately sized packets.

4.4.2 Write command

The function `send_ping()` provides the user with the ability to send two pings to the device for processing. The two pings are passed to the function, which then breaks each 32-bit value up into bytes to be sent over USB and inserts them into a buffer interspersed with packet headers as specified in section 4.2.2.5. For two pings of 256 samples 64 packets of 35 bytes must be sent, so a buffer of 2240 bytes is required. The host then sends status requests to the USB device and reads from the IN endpoint until it receives a reply to say that both ping buffers are empty. The entire buffer is then written to the device's OUT endpoint using the libusb function `usb_bulk_write()` and the device processes the packets as described in section 4.2.2.5.

4.4.3 Read command

The function `get_ping()` allows the user to read processed data back from the device. Two output arrays are passed to the function, the first is unsigned ints and the second is unsigned shorts, both are 256 entries long. The function sends a buffer of read headers to the device as a bulk write operation, then executes a bulk read of length 1536; this is the number of bytes that should be returned for the read headers that were sent.

The timeout on the bulk read has to be set long enough to give the device time to send all the data, as it is written to the endpoint in small sections during each iteration of the correlator. Once the read buffer is full, the bytes are pieced together into integers and shorts and written into the output arrays.

4.4.4 Fine delay estimation

The fine delay estimation requires an inverse tan operation to calculate the phase of the correlation result. C Library functions provided for this purpose operate exclusively on floating point numbers so to implement fine delay estimation on the FPGA would require the development of a fixed-point routine or that a floating point unit be synthesised as part of the MicroBlaze. Since this would require additional device area, and the delay estimation is not particularly computationally intensive, it was decided to implement the algorithm in floating-point on the host PC. After a processed ping has been read back from the FPGA, the results are passed to the fine delay algorithm, which operates in the following manner. For each pixel in the ping:

1. The real and imaginary correlation results and the coarse delay in samples are converted from 16-bit fixed-point format to double precision floating-point.
2. The phase of the correlation result ϕ is found using the library routine `atan2()` which returns a result between $-\pi$ and π.
3. The coarse delay is found by simply multiplying the number of samples by the sample spacing.
4. An estimate of the fine delay is made by applying (4.20), where k is the number of cycles from the start of the window calculated as the coarse delay multiplied by the centre frequency of the signal. Because of the low accuracy of the coarse delay, this parameter is subject to an ambiguity that must be resolved to find the correct delay.

$$\hat{\tau}_f = \frac{-\phi}{2\pi f_c} + \frac{k}{f_c} \tag{4.20}$$

4.4.4.1 Resolving phase ambiguity

Because of the wrapping property of the phase measurement and the level of accuracy of coarse delay measurements, it is possible that the value of k as calculated above is incorrect by one cycle if the angle is close to the $-\pi/\pi$ branch cut. In the simplest method to correct for the ambiguity, the fine delay is evaluated with three values of k about the calculated value; the value of k that minimises the difference between the fine and coarse delay estimates is taken to be the correct one. This is rather computationally

complex as the fine delay equation is evaluated three times to garner one result, so a more elegant solution is desirable.

One such method resolves the ambiguity without excessive calculation by rotating the vector when the phase is close to the branch cut, as shown in Figure 4.13 below.

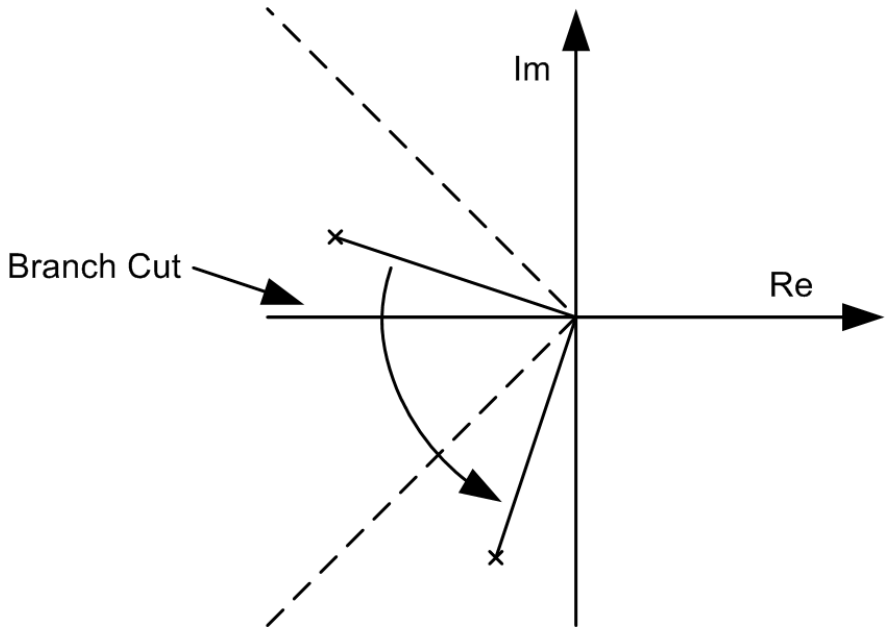


Figure 4.13 Rotation of correlation result away from $\pi/-\pi$ branch cut.

If the vector is within $\frac{\pi}{4}$ of the branch cut, the vector is rotated away from the branch cut by $\frac{\pi}{2}$ and the phase recalculated. The number of cycles is then calculated as:

$$k = \text{round}(f_0\tau_c - 0.25) + 0.25. \tag{4.21}$$

The fine delay can then be estimated with (4.20) in the normal manner.

4.5 SUMMARY

This implementation is valid as a proof-of-concept for a real-time FPGA-based bathymetry coprocessor operating over a USB 1.1 interface. In the next chapter, the performance of each component in the system is examined for its suitability in a fully functional system and recommendations for potential improvements are made where necessary.

Chapter 5

RESULTS

The system described in the previous chapter proves the concept of a real-time FPGA-based bathymetric processor, but the architecture requires some improvement to become a complete solution. In this chapter, the implementations described in Chapter 4 are discussed in terms of their speed, accuracy, the device area they occupy, and the extent to which they satisfy the requirements of a complete real-time solution. The bathymetry generated for a simulated scene is also presented.

5.1 COPROCESSOR

In the current configuration, the bathymetry coprocessor operating on the FSL does the majority of the processing and is the bottleneck in the system. The accuracy of the results is slightly lower than can be achieved on a PC, due to the fixed-point implementation.

5.1.1 Execution time

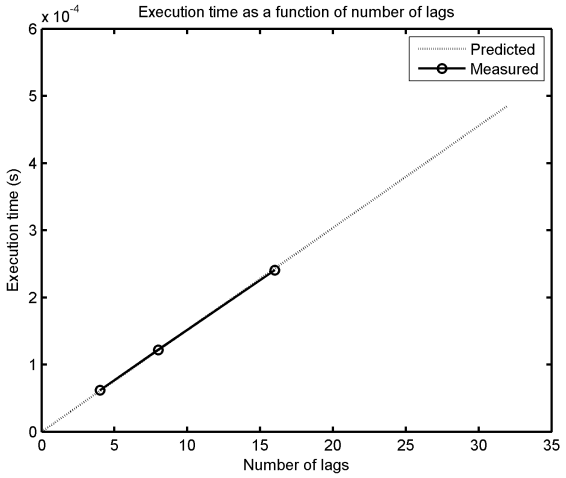
The speed of the coprocessor is defined by four limiting factors: the clock speed, the architecture, i.e., parallel or serial, the size of the correlation window, and the number of correlation lags. The differences between parallel and serial implementations were discussed in section 4.3.3; in terms of speed, the proposed parallel design would offer approximately a fourfold increase over the serial design.

For the serial design that was implemented, the execution time is defined approximately by (5.2) below:

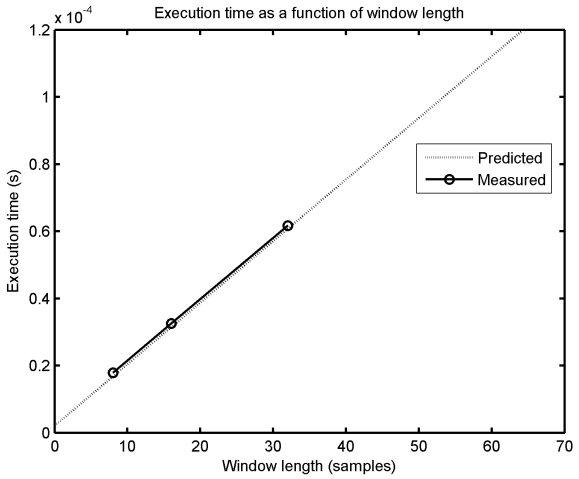
$$t_{\text{ex}} = \frac{\text{Num\_lags} \times (\text{Window\_len}(4 \times (\text{Fetch/Mult/Acc})) + \text{Peak\_detect})}{\text{Clock\_freq}} \quad (5.1)$$

$$= \frac{\text{Num\_lags} \times (\text{Window\_len} \times 22 + 20)}{48\,\text{MHz}} \quad (5.2)$$

The parallel design improves upon this by executing the four fetch, multiply and accumulate cycles in parallel. In the serial implementation, the window length and number of lags are the main parameters affecting the execution. Figures 5.1(a) and 5.1(b) show the effect that window length and lag have on execution time, both theoretically as derived in (5.2) and practically on the FPGA.



(a) Lag varied for window length of 32.

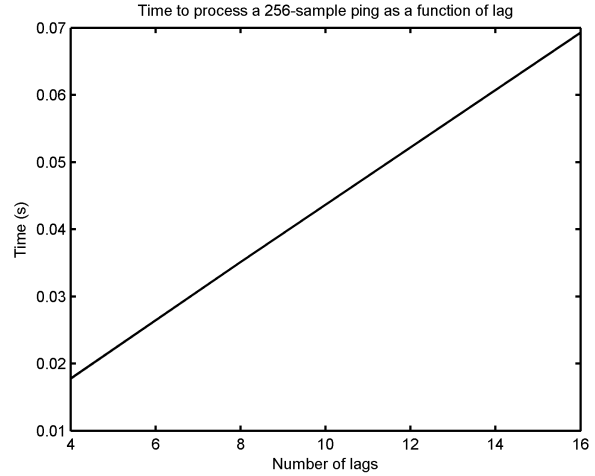


(b) Window length varied for lag of 4.

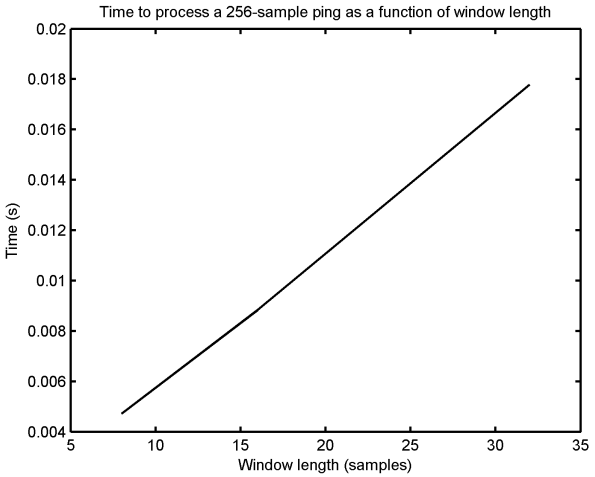
Figure 5.1 Shows the effect on execution time of varying the lag and window length of the correlation. In (a) the window length is held constant at 32 samples while the lag is varied between 4, 8 and 16. In (b) the lag is held constant at 4 while the window length is varied between 8, 16 and 32 samples.

Evidently the practical results follow the theoretical projections closely. Since the coprocessor is the bottleneck in the system, the total execution time required to process a ping can be inferred from these results. This is shown in Figures 5.2(a) and 5.2(b) below.

It is desirable to reduce the execution time of the system such that the USB 1.1



(a) Lag varied for window length of 32.



(b) Window length varied for lag of 4.

Figure 5.2 Shows the effect of varying the lag and window length on the time to process a 256-sample ping. In (a) the window length is held constant at 32 samples while the lag is varied between 4, 8 and 16. In (b) the lag is held constant at 4 while the window length is varied between 8, 16 and 32 samples. These values are simply inferred from the measurements in Figures 5.1(a) and 5.1(b).

link becomes the bottleneck in the system. From the above figures it can be seen that the fastest execution time occurs with a window length of 8 and 4 lags. The maximum delay between the two receivers is 3 samples, as calculated below. Therefore the number of lags can safely be reduced to 5, allowing for the values either side of the maximum to be calculated, as required for quadratic fitting.

$$\begin{aligned}
\text{Max Delay} &= \frac{\text{Separation of top and bottom receiver}}{c} && (5.3)\\
&= \frac{0.15\,\text{m}}{1500\,\text{m/s}} && (5.4)\\
&= 100\,\mu\text{s} && (5.5)\\
\text{Max Samples} &= \frac{\text{Max Delay}}{\text{Sample Spacing}} && (5.6)\\
&= \frac{100\,\mu\text{s}}{33.3\,\mu\text{s}} && (5.7)\\
&= 3\,\text{samples} && (5.8)
\end{aligned}$$

Reducing the window length to 8 will reduce the quality of the image produced. A larger window length increases T in the CRLB (2.39) — decreasing the variance — while at the same time decreasing the resolution of the result. Therefore, a window length of 8 will yield a higher resolution image, but each pixel will be less accurate due to the decreased time extent of the input windows. Such degradation in image quality is unacceptable, thus window length should remain at 32.

The final variable affecting the execution time is the clock speed. Currently the coprocessor is clocked at 48 MHz, the same rate as the MicroBlaze and the rest of the system. As the FSL interface supports asynchronous clock domains, the coprocessor could run on a faster clock, decreasing the execution time. Synthesis reports indicate that the coprocessor module could be clocked at up to 150 MHz, yielding a threefold improvement in execution time.

5.1.2 Accuracy

Because this is a real-time application, it is not crucial that images are produced with the same accuracy as methods that are not time critical, as the results will only be displayed on screen for a matter of seconds. If areas of interest are found in the real-time images, they can be subjected to more intensive processing at a later time. However, it is still desirable for the system to produce the most accurate results possible, within the time constraints.

The correlator was tested to ensure that the correct results were computed. Results can be difficult to verify with typical SAS ping inputs, so initially the correlator was tested with two square wave inputs, one delayed by 4 samples. The input and output

waveforms are shown in Figure 5.3 below. As expected, the correlator found the delay between the waveforms to be 4 samples, and correctly calculated the magnitude as verified in the equations below.

The correlation will be at a maximum when the two waveforms are lined up, so

$$\text{Corr}_{\max} = \text{Amplitude}^2 \times \text{Length} \tag{5.9}$$

Because of the fixed point implementation, the amplitude is 32767, but here it is assumed to be 32768 for simplicity

$$\begin{aligned} \text{Corr}_{\max} &= \left(2^{15}\right)^2 \times 2^3 && (5.10)\\ &= 2^{33} && (5.11) \end{aligned}$$

This result is in Q30 format, so is divided by 2^{15} to convert back to Q15

$$\begin{aligned} \text{Corr}_{\max} &= \frac{2^{33}}{2^{15}} && (5.12)\\ &= 2^{18} && (5.13) \end{aligned}$$

The accumulator has 6 guard bits to prevent overflow, so all results are scaled down by a further factor of 2^6

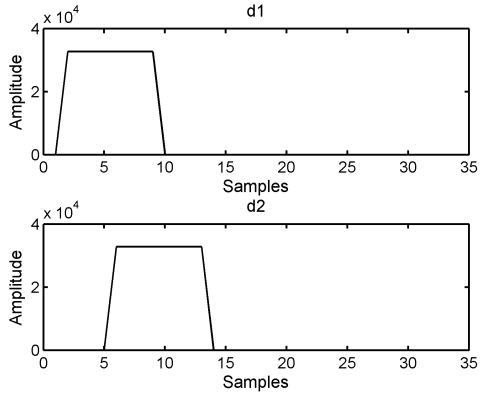
$$\begin{aligned} \text{Corr}_{\max} &= \frac{2^{18}}{2^6} && (5.14)\\ &= 2^{12} && (5.15)\\ &= 4096 && (5.16) \end{aligned}$$

This complies with the result output by the correlator of 4095; the small discrepancy is introduced by the assumption made in (5.10).

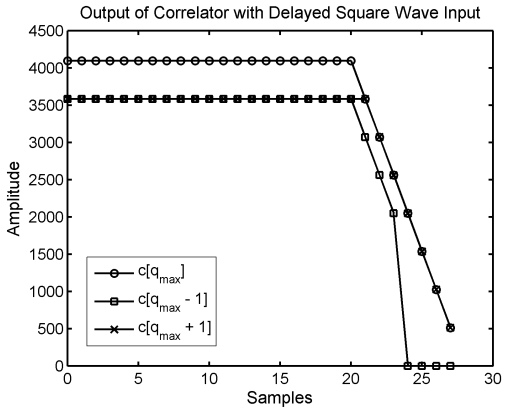
Since the delay between the two input windows is rarely an exact number of samples, the correlator was also tested with two triangular waves, offset by 2.5 samples as shown in Figure 5.1.2. The half-sample offset was obtained by up-sampling the original waveform by two, shifting by one sample then downsampling to the original rate. It can be seen in Figure 5.1.2 that the resulting delay is still a discrete value — the exact delay is found by fitting a curve to the points produced by the coprocessor. This is carried out on the MicroBlaze processor, and the results described in Section 5.2

5.1.2.1 Bias

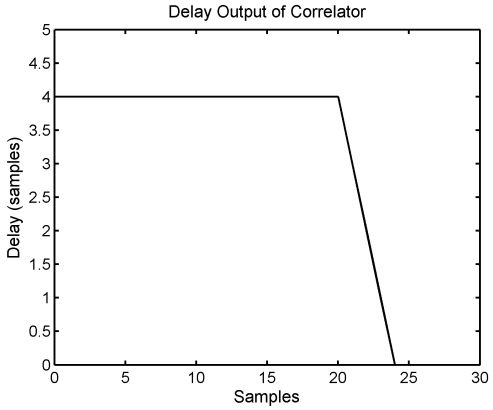
The correlation calculated by this method is biased because as one signal is moved past the other the index for d_2 moves outside the window, in which case zeros are substituted. This means that as each lag is calculated the potential maximum is reduced, possibly



(a) Input waveforms.

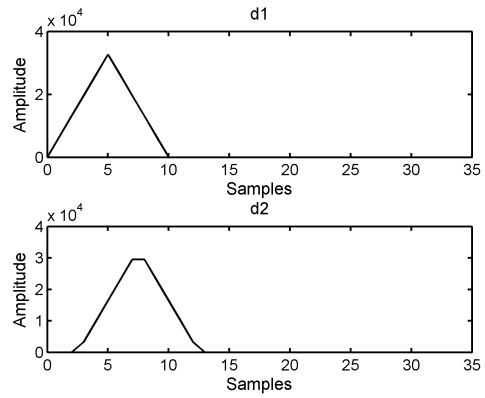


(b) Correlator output.

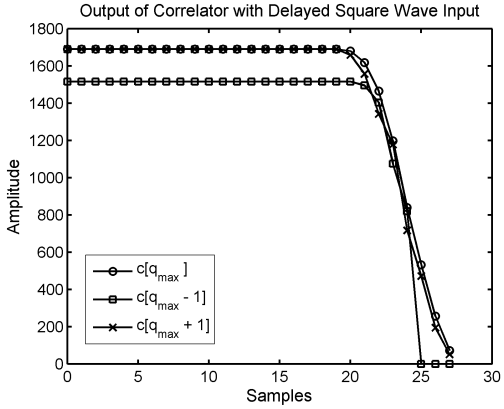


(c) Calculated Delay.

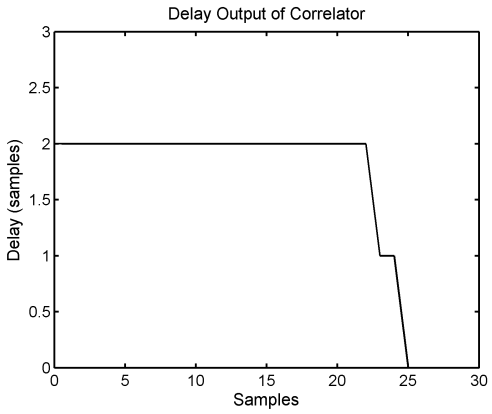
Figure 5.3 The correlator was passed the two input waveforms in (a) and generated the output in (b), as well as correctly finding the delay between the two signals (c). Note that the results decay as the signal begins to move out of the window.



(a) Input waveforms.



(b) Correlator output.



(c) Calculated Delay.

Figure 5.4 The correlator was passed the two input waveforms in (a) and generated the output in (b). The correlator finds the delay as 2 samples as only discrete values are calculated. However, the sample to the right of the maximum is equal to the maximum as shown in (b), which results in a delay of 2.5 samples calculated in the quadratic fitting stage. Note that again the results decay as the signal begins to move out of the window.

resulting in the maximum being found at the wrong lag. To un-bias the correlation, two solutions are possible: either each result can be scaled up or d_2 can be implemented with a "sliding" window, whereby the data is windowed after the lag is set. From a hardware implementation perspective the latter is the easier solution, as it merely entails keeping a larger buffer for d_2 to account for the shifting window. The former requires a division operation, the implementation of which is much more involved [44].

5.1.3 Device area

The device area used by the coprocessor with a 16-lag, 32-sample window is 532 slices. This equates to approximately 10% of the device area in the 2VP7 device. The MicroBlaze is capable of supporting 8 FSL links so in theory 8 coprocessor blocks could run in parallel, but the current size prohibits this. The infrastructure on the chip excluding the coprocessor occupies approximately 40% of the device, so the coprocessor would have to be reduced in size to fit the maximum number possible. Reducing the lags and the window size has little effect on the size of the coprocessor.

Also to be taken into consideration are the FPGA's hard-wired resources: BRAMs, multipliers and the PowerPC core. In a serial configuration the coprocessor uses one BRAM and one 18x18 multiplier block from the 44 of each available. A parallel implementation would use 2 BRAMs and 4 multipliers, so 8 such modules along with the MicroBlaze using 16 BRAMs for internal memory and the USB module using 6 BRAMs for FIFOs would use 86% of the available BRAM resources. Since the PowerPC core is embedded in the silicon of the FPGA, substituting it for the MicroBlaze processor would release more of the device area to be used for correlators. The correlator interface would have to be modified to connect to the PLB rather than the FSL, but this would have the benefit of removing the limit of 8 coprocessors, allowing as many as would fit in the device.

5.2 MICROBLAZE

As described in Chapter 4, the main tasks of the MicroBlaze processor are buffering data from the USB interface and finding the exact location of the correlation peak from the estimate given by the coprocessor. These steps are performed in parallel with the coprocessor.

5.2.1 Interpolation

Since the correlator only returns the three points about the peak, there is only enough information to apply either a quadratic or a linear interpolation scheme. For the case of finding the peak in the magnitude signal, linear interpolation will not increase the accuracy at all, so quadratic fitting is used to generate an equation for the curve which

is then differentiated and solved to find the maximum. With a clock speed of 48 MHz, this process takes approximately 1.5 μs.

Once the exact location of the peak in magnitude has been found, the values of the real and imaginary components of the signal must be interpolated at that location. Unlike the peak finding operation, this can be done using either linear or quadratic methods, resulting in a tradeoff between accuracy and speed.

5.2.1.1 Comparison of linear and quadratic interpolation

In order to ascertain the interpolation effect in the SAS system, the two interpolation schemes were tested on a typical correlation, shown in Figure 5.5. The correlation was first oversampled by a factor of 150, then interpolation was performed between three samples separated by 100, using both quadratic and linear methods. Values ranging from -0.5 to 0.5 of a sample either side of the middle sample were calculated, then compared with the actual sample at that point to ascertain the error. The process was then repeated on another three samples until the entire waveform had been covered. This is equivalent to interpolating between samples with an oversampling factor of 1.5, as used in the KiwiSAS system. The Mean Squared Error (MSE) was then calculated for each delay from -0.5 to 0.5, yielding the results in Figure 5.6. As expected, both methods give the worst results halfway between samples, but the quadratic method does offer some improvement over linear interpolation. Further analysis of the effect of oversampling on interpolation schemes can be found in [7, pp. 60–65].

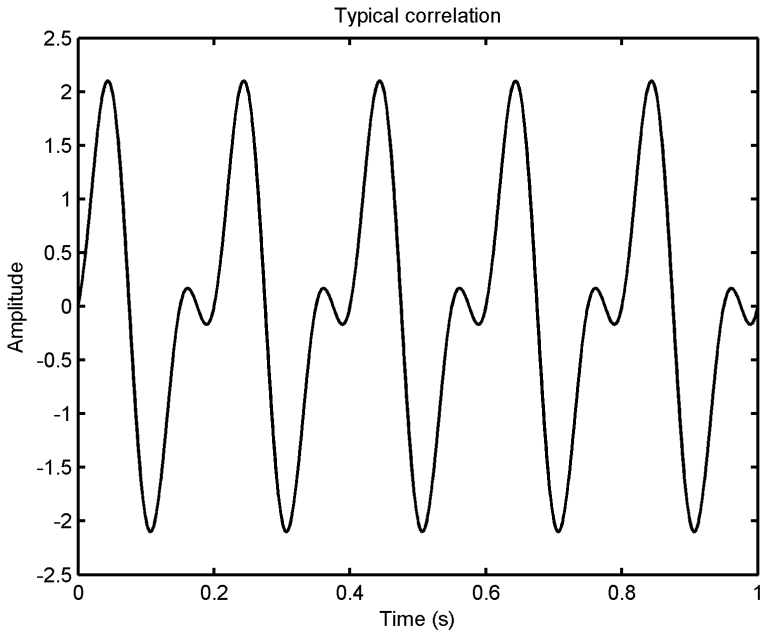


Figure 5.5 A typical correlation, albeit at a very low frequency. The waveform comprises two sinusoids at 5 Hz and one at 10 Hz and is sampled at 30 Hz.

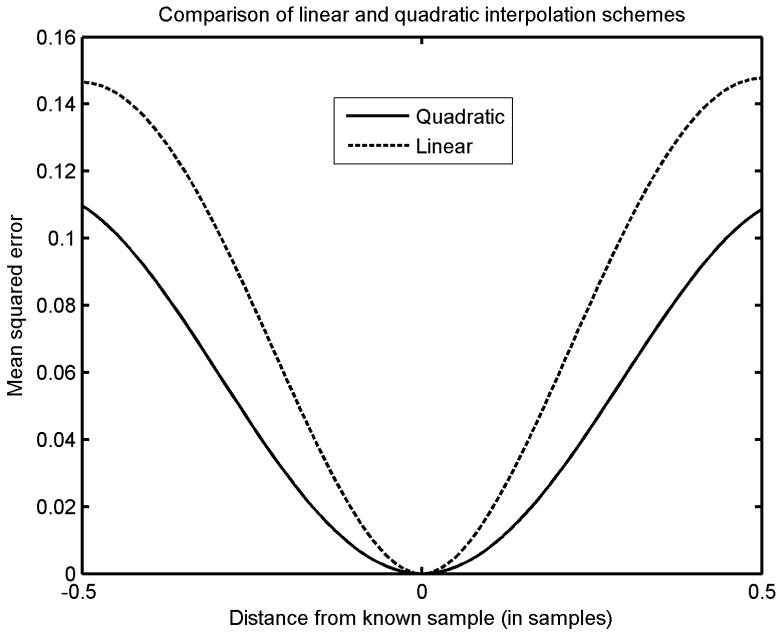


Figure 5.6 The MSE of each interpolation scheme for an oversampling rate of 1.5 — the quadratic method shows some improvement over the linear. The advantage increases for higher oversampling rates.

The extra accuracy afforded by quadratic interpolation comes at the cost of execution time however. The quadratic interpolation algorithm is approximately twice as computationally intensive as the linear algorithm, taking 7.2 μs compared with 3.6 μs for linear. In the project's current configuration this is of little consequence, as there is a large amount of slack between the MicroBlaze and the coprocessor execution times. If the design was expanded to include more coprocessors the execution time would become more significant, as more interpolation operations would need to be done sequentially within the same time limit.

5.2.2 Normalisation

The normalisation routine was applied to the correlator output from the above tests. In the case of the square wave input the output is normalised to 16-bit full scale as the two inputs are identical, and the triangular wave is normalised to just below full scale due to the difference in sampling between the two. The normalisation routine takes 12.8 μs to execute which adds significantly to the total execution time per correlation cycle, but at 52 μs total per sample the MicroBlaze code is still well within the limits for a real-time solution and significantly faster than the correlator.

5.2.3 Data transfer

The other operation taking place in parallel with the coprocessor is the transfer of raw and processed data over the USB interface. The outcomes of the hardware implementation are discussed in detail in section 5.3, but the software routines are analysed here, as they are responsible for a large proportion of the total MicroBlaze execution time.

5.2.3.1 Interfacing to USB

Once during every correlation cycle, a function is called to transfer data from the USB function core into the MicroBlaze internal memory. The data transfers are kept small so as not to delay the operation of the coprocessor. If there is no data to be transferred to the USB core the function will exit immediately, taking an insignificant amount of time. Otherwise, either a data transfer of 35 bytes or a status transfer of 6 bytes will occur, requiring 29.6 μs and 2μs respectively. The time for a data transfer can be reduced to 24 μs by unrolling loops in the transfer function, as this reduces the access time to the USB core from 14 cycles to 7 cycles. Writing to a register in the OPB GPIO peripheral takes 6 cycles, so USB access times are in line with the expected level of performance for a register interface. During a data transfer, the execution time is split approximately evenly between transfers to the USB core and assembling the received bytes into 32-bit integers to be stored in memory. As shown in (5.18) below, a transfer of 35 bytes in 24 μs equates to a data rate of 11.6 Mbps.

$$\text{Data rate} = \frac{\text{Number of Bytes Transferred} \times 8}{\text{Transfer Time}} \tag{5.17}$$

$$= \frac{35 \times 8}{24\,\mu\text{s}} \tag{5.18}$$

$$= 11.66\,\text{Mbps}$$

However, data transfer is only active for a sixth of the correlation cycle, so the actual data rate as seen from the host side is a lot lower. In a fully optimised system data would be transferred between the MicroBlaze and the USB core continuously, thus utilising the full bandwidth of the USB 1.1 interface.

5.2.3.2 Memory requirements

Due to the small ping length currently implemented the ping buffers are small enough to reside in internal memory, requiring a heap size of 12 kB. After including 12 kB for the stack and approximately 8 kB for the program itself, the 32 kB internal memory is almost full. Ideally, the ping length would be increased to 2048 or 4096 samples, and up to ten pings would be processed simultaneously, allowing the correlation results

to be averaged across pings for increased accuracy. This would require the system to process data at a rate of approximately 39 Mbps.

Increasing the buffering on the chip to meet these demands increases the system's memory requirements proportionally. The internal memory available can be increased by decreasing the size of the stack — 12 kB is an overly pessimistic estimate of the stack size requirement — and by increasing the size of the internal memory to 64 kB, the maximum allowable. This would allow the data set to be expanded by up to 4 times, but would place a high demand on the BRAM resources in the chip, leaving only 5 spare. The number of BRAMs required to store the extra data would limit the number of additional coprocessors that could be implemented to 5 in a serial implementation or 2 in a parallel implementation, thus defeating the purpose of expanding the data set. Additionally, the storage demands may increase by up to 160 times over the current system, so increasing the addressable memory by 4 times is insufficient. The solution in this case is to utilise external memory.

On the Virtex-II pro board 32 MB of SDRAM is provided. This is more than adequate for the application, as the worst case described above requires just under 2 MB of memory. The major design problem in using external memory is the increased latency of accessing a separate chip, but as discussed in Chapter 3 most of these inadequacies can be overcome through the use of caches, burst transactions and DMA.

5.3 USB 1.1

The USB 1.1 protocol was used to transfer data between the host PC and the board. In this section the performance of the USB core is discussed, and the USB 1.1 protocol is compared with other interfaces for use in an optimised system.

5.3.1 OPB interface

The OPB interface between the USB core and the MicroBlaze performs adequately for this application, delivering access times of 7 cycles per byte, which is equivalent to 54 Mbps. Since this is 4.5 times faster than the maximum data rate for USB 1.1 the system will easily be able to keep up with the data demands of the host PC. The FSMs used to transfer data between the software-accessible registers and the endpoint FIFOs of the USB core can only operate while the registers are not being accessed, as allowing this could result in data being lost. Potentially data could be lost if two read or write transfers were issued in quick succession but in practice this is not an issue. The transfer between the register and FIFO is a single-cycle process, which has time to occur while the MicroBlaze stores the result of the USB access.

The bandwidth of the USB OPB link could be expanded by implementing burst transfers and DMA. In the case of burst transfers the method for interfacing between

the OPB IPIF and the USB core would need to be altered, as there would not be enough time between the individual bytes of a burst to write register contents to the endpoint FIFOs. The best way of implementing a burst system would be to use a small FIFO to buffer data between the IPIF and the USB core. The FIFO interface provided within the IPIF core would be ideal, but the fixed width of 32 bits means the solution would be somewhat inefficient. However, this would be a fair tradeoff for the extra performance leveraged. Combining burst transfer capabilities with a DMA setup would further improve the performance of the system, especially if external memory was used.

5.3.2 Host

Arguably the most important aspect of the design is the effective data rate from the host PC. The design can process data at a rate of 15 256-sample pings per second, or approximately 250 kbps, but this is limited by the speed of the coprocessor rather than the USB link. To test the speed of the USB the coprocessor functionality is removed and a large number of bytes are written. Due to the large overhead of the libusb bulk transfer functions, transfers become more efficient as the size increases — the relationship between size and data rate is shown in Figure 5.7 below.

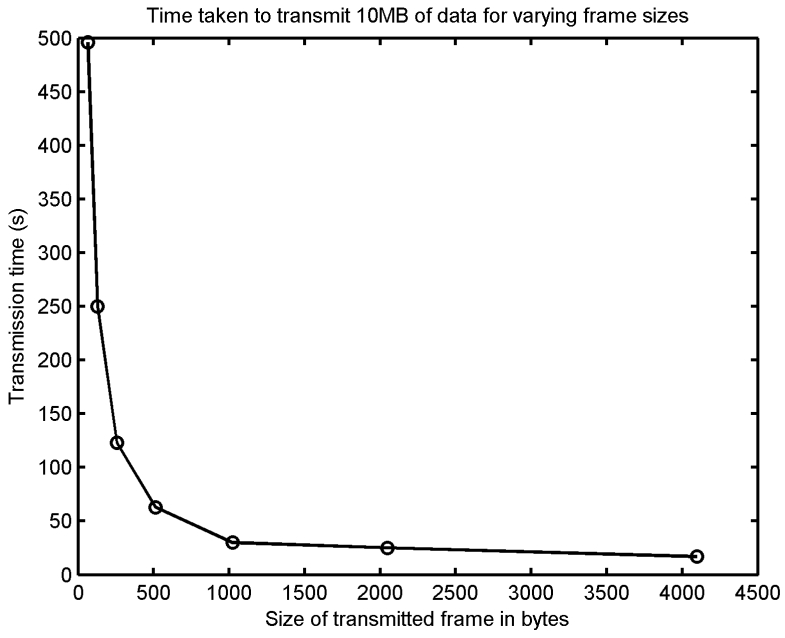


Figure 5.7 Time taken to write 10 MB of data over the USB 1.1 link using varying frame sizes.

Evidently, the best system performance will be achieved when using large data transfers, and using a 4 kB frame size a transfer rate of 4.7 Mbps is achieved. However this requires the use of large FIFOs to guarantee that the data will be buffered at the device end. In the current device 4 kB per endpoint is the largest practical FIFO

size, using 8 BRAMs for two endpoints. Larger FIFOs would require too large a percentage of the BRAMs, as they are also required for MicroBlaze internal memory and coprocessor memory. Also, it is evident from Figure 5.7 that performance gains from FIFOs larger than 4 kB would be minimal, as the speed seems to be limited by the USB API on the host. However, this issue highlights the BRAMs as the critical resource in the system; the performance of all the major system components depends on their availability. Since they are a hard-wired resource the design can only be expanded beyond the bounds defined by the BRAM resources by migrating to a larger FPGA.

5.3.3 Reliability

The USB interface was tested for reliability by writing 100 MB of data to the FPGA where the value of each byte was incremented before being sent back to the host. At the host, the data read back from the FPGA was compared with the expected results for errors. Over 100 MB, not a single byte was returned incorrectly, so clearly the reliability of the USB implementation is more than adequate for the application.

However, the race condition discussed in Chapter 4 is a major problem for the stability of the design. When changes are made to the hardware platform the USB core frequently ceases to work, requiring a process of tweaking the place and route options until a combination is found to give a working design. It is thought that the accuracy of the clock is the cause of the race condition, as the USB specification [37] requires an accurate 48 MHz clock with a level of jitter not exceeding 0.5%. Clock jitter can be defined as the variation in clock period between cycles, i.e., the amount of time that the actual clock edge lags or leads its expected position.

The crystal provided on the development board is 100 MHz, so a Digital Clock Manager (DCM) block is used to reduce the clock speed to 48 MHz by dividing by 25 and multiplying by 12. Generating a clock in this manner uses the frequency synthesis mode of the DCM rather than the Delay Locked Loop (DLL) used when the clock required is a straight multiple of the input clock. Frequency synthesis mode does not use a feedback path to compare the output clock with the input, and as such is subject to more jitter than the DLL. The jitter of the 48 MHz frequency synthesis generated clock was calculated using the Xilinx Virtex-II jitter calculator [45] to be 6.6%, far outside the bounds imposed by the USB specification. This may explain why the core frequently is frequently inoperational. Occasionally a certain combination of place and route parameters will result in a floorplan design that is tolerant of this level of jitter, producing a working system, but this is not an adequate solution to the problem.

To solve the problem, an alternative to the 100 MHz crystal must be found for the clock source. The development board includes a port for connection of an external clock source, where a 24 or 48 MHz clock could be connected, allowing the DCM to operate in DLL mode rather than as a frequency synthesiser. If the design was implemented

on a custom PCB a 24 MHz or 48 MHz crystal could be incorporated directly on the board. Hopefully these changes would increase the accuracy of the clock and hence the stability of the design.

5.3.4 Suitability

The minimum data rate requirement of the current system is approximately 500 kbps, and as shown in the previous section the USB can provide a data rate of 4.7 Mbps, more than enough to meet the demands of the system in real-time. The bandwidth requirement of the fully functional system described in Section 5.2.3.2 is approximately 70 Mbps, covering transfers to and from the FPGA. This is outside the bounds of what USB 1.1 can provide, so a more advanced communication protocol will be required in the final system.

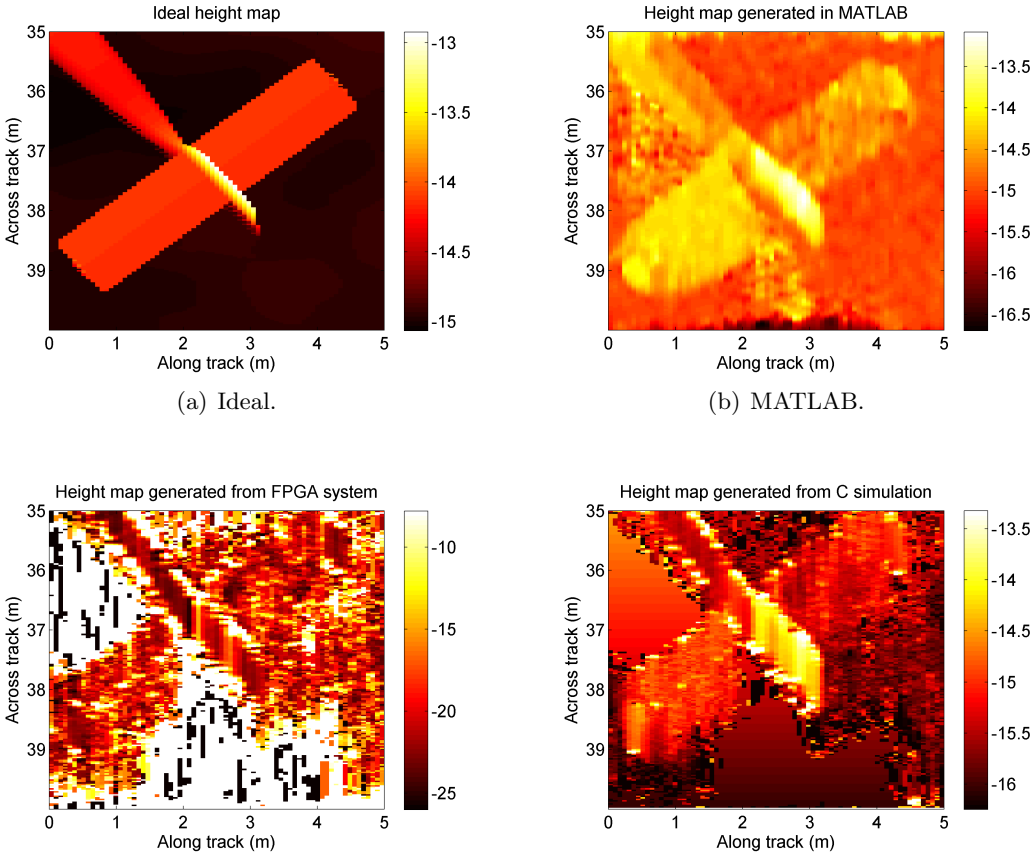
USB 2.0 is the obvious choice as it is supported at the host with the same libusb API as USB 1.1, and the operation of the core is also similar. Upgrading the USB would expand the bandwidth to a potential 480 Mbps, although these speeds are difficult to reach in practice due to the overheads associated with each transfer. The main disadvantage of USB 2.0 is the requirement for an external PHY to handle the high clock speeds necessary for high-speed communications. Therefore to implement USB 2.0 a daughterboard would have to be designed to connect the PHY to the FPGA. Additionally, the USB 2.0 core is more complex than USB 1.1 and hence requires more device area.

Fast Ethernet is another option, offering 100 Mbps of bandwidth and the advantage of compatibility with the communications scheme already in place on the towfish. No extra external hardware is required to implement Ethernet, as the necessary components are already present on the P160 communications expansion board used for USB 1.1. A 10/100 Ethernet core is provided with the EDK package, reducing the development time to implement the protocol. The disadvantage of Ethernet is that it requires some type of operating system to handle the TCP/IP stack, adding overheads to the MicroBlaze code. The PowerPC core is better suited running an OS as it includes a dedicated MMU and has higher performance in general, so Ethernet could be a viable option if the project was migrated to PowerPC.

Finally, if a PCB was designed for the system a PCI interface could be used, allowing the board to plug into the towfish or a PC on the tow vessel. A PCI IP core is included with the EDK package for use on the OPB, and provides a fully functional 32-bit bridge to the PCI bus, compliant with Revision 2.2 of the PCI bus standard.

5.4 BATHYMETRIC RESULTS

The system was tested with data generated by the alSAS simulator [8]. The scene shows the tail section of a partially buried "Stuka" aeroplane. The four images below show the ideal height map, the height map processed in MATLAB with a maximum likelihood estimator, on the FPGA, and in C software emulating the FPGA processor. Clearly, the image generated by the FPGA is of much poorer quality than that processed in MATLAB, yet the C simulation gives results of similar quality to the MATLAB routines. There is a noticeable difference between the C and MATLAB results because the ML estimator achieves a lower CRLB (see section 2.2.1), and the C model uses fixed-point arithmetic instead of floating-point. The discrepancy between the FPGA and C simulation results is caused by an oversight in the quadratic fitting routine used to obtain the coarse peak estimate, discussed in section 5.4.1.



(a) Ideal. (b) MATLAB.

(c) FPGA. (d) C simulation.

Figure 5.8 The ideal height map of the Stuka tail section is shown in (a), and a MATLAB reconstruction using a maximum likelihood estimator in (b). The FPGA processed image in (c) should be similar to (b), but is drastically worse because of an unforeseen error in the code. A C simulation of the FPGA process was run with the error removed and generated similar results to the MATLAB routines (d), although the image quality is noticeably worse because the ML estimator achieves a lower CRLB (see section 2.2.1).

5.4.1 Coarse peak estimation issue

The quadratic peak finding routine currently uses 32-bit fixed-point integers, so the denominator is scaled down as described in section 4.2.2.2 to set the precision of the output, as scaling up the numerator will certainly lead to overflow in some cases. While this produces the correct results in test cases such as those in section 5.1.2, the typically smaller correlation results generated from the simulated data are scaled to zero in the majority of cases. Dividing by zero then gives an unexpected result, and consequently the coarse delay estimates generated are wildly inaccurate. This problem was corrected in the C simulation by using floating-point arithmetic for the peak estimation routine, but ideally fixed-point arithmetic would be used throughout the embedded system. As the maximum precision available from the MicroBlaze processor is already being used, the most plausible solution to this problem is to implement the peak finding scheme in hardware as part of the coprocessor. This would allow the design of a system where the numerator is scaled up rather than the denominator being scaled down, hence avoiding the divide-by-zero problem.

5.4.2 Using coherence map to improve images

The height map can be further improved by utilising the coherence map generated by the normalisation process. Taking the absolute value of the complex normalised correlation result gives a measure of the significance of the delay calculated for each resolution cell, ranging between 0 (signals are completely uncorrelated) and 1 (signals are identical). The coherence map generated for the test scene is shown in Figure 5.9. The shadow regions seen in Figure 5.8 above are observed to have a coherence of zero, which is expected as the towfish does not receive any echo from those areas. By thresholding the coherence map at a certain level, the height map can be improved by discarding pixels with poor coherence [7, Chapter 3].

The coherence map in Figure 5.9 is thresholded to 0.65 and then applied to the height map, setting any resolution cell deemed not significant to the floor of the height map. The height values in these cells are then interpolated from neighbouring cells in the across-track dimension to give the final image shown in Figure 5.10, clearly improved from original generated with raw delay values.

5.5 OPTIMISATION

This section presents an outline of the changes that are required to optimise the system, giving consideration to the resources available on the current chip, as well as discussing the improvements that could be possible with a high-end FPGA.

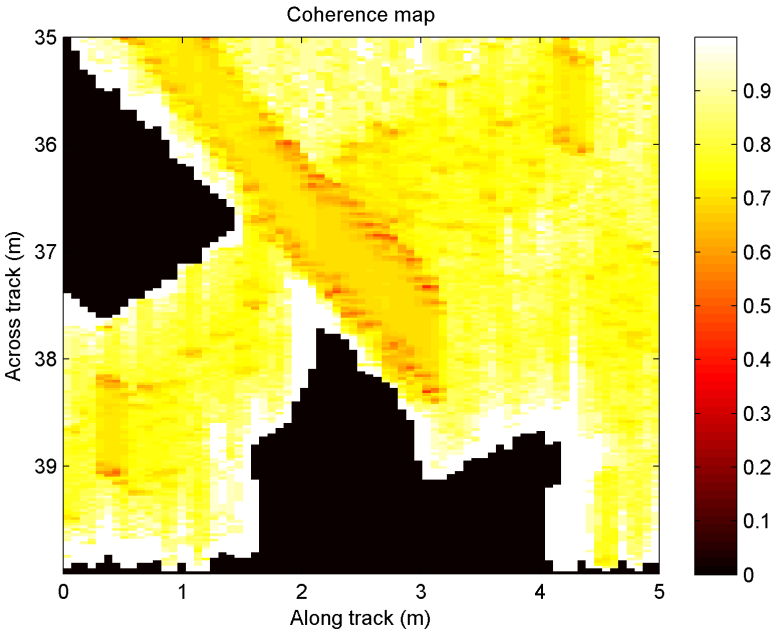


Figure 5.9 Coherence map generated for Stuka tail section.

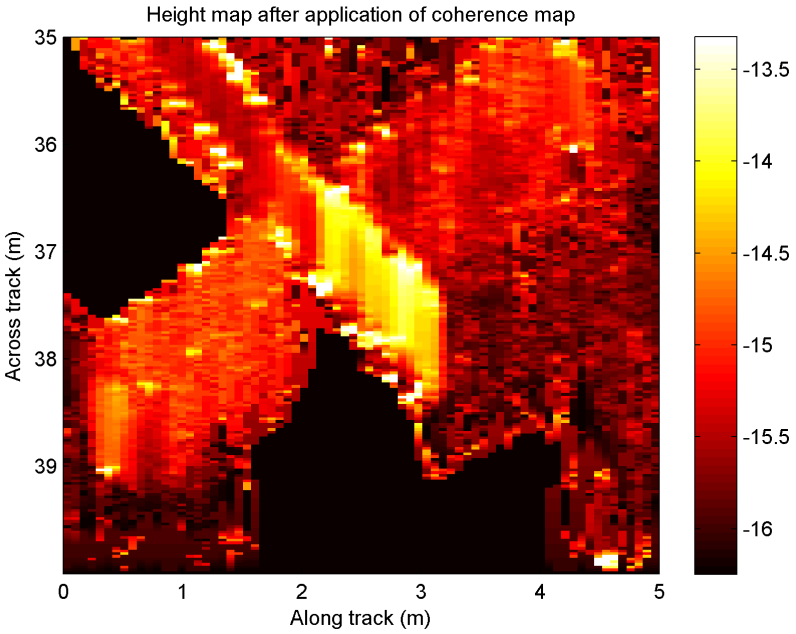


Figure 5.10 Height map after application of coherence map.

5.5.1 Improvements to current system

Without altering the IO interface or changing the FPGA device, there are a number of design changes that could be made to optimise the system Firstly, the number of lags computed by the correlator could be reduced from 16 to 5 without affecting the accuracy of the system, speeding up the correlation by a factor of three. Secondly, the correlator design should be modified to a parallel implementation to further increase performance and make use of the ample hardware multiplier resources that are currently unused. To take advantage of the increased speed, the internal memory should be increased to 64 kB, allowing 3 times the data to be buffered. Finally, as much of the remaining device area as possible should be filled with correlator blocks, yielding a design with 3 coprocessors working in parallel with the MicroBlaze. Making these changes would give an estimated data rate of 3.8 Mbps, which is close to maximising the available USB 1.1 bandwidth.

5.5.2 Fully functional system

With a higher-bandwidth IO solution such as USB 2.0 or Ethernet, more significant changes would be made to the system. The PowerPC microprocessor could be substituted for the MicroBlaze, giving higher performance and freeing up more device area for use as correlator blocks. This would require that the correlator block be wrapped for the PLB, as the FSL is not available on the PowerPC core. External memory would be used for storing data as internal memory is too limited in size, and the communications core would need DMA capabilities to allow efficient data transfers. This design would require approximately 1400 slices or 28% of the device area for the USB 2.0 core and 7% for the memory controller, leaving approximately 60% of the device available for correlator blocks, so up to 6 could be implemented. If the correlators were fully optimised this would give a processing bandwidth of approximately 57.6 Mbps, more than enough to meet the data rate specified in Section 5.2.3.2.

Increasing the number of correlators increases the amount of time that the microprocessor spends administrating the data, which is why a DMA scheme is vital in this system. As shown in section 5.2, the MicroBlaze spends approximately 60% of its execution time transferring data from the USB, so allowing the USB core itself to write to memory would greatly speed up the processor code.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

A functional real-time bathymetric SAS processor has been implemented on a Virtex-II Pro ™ FPGA development board and a driver for the system developed for the Linux operating system. The system is a proof-of-concept rather than a fully functional design; to process the entire bathymetric scene requires a high data transfer rate between the host PC and the FPGA, and the use of USB 1.1 has restricted the rate at which data can be processed.

The key components in the design are a USB 1.1 interface core, a bathymetry coprocessor, a MicroBlaze processor core and a driver on the host PC. Custom hardware was developed in VHDL for the coprocessor and for wrapping an open source USB core to be compatible with the rest of the system. C code was developed for administration tasks on the MicroBlaze embedded processor and to provide a simple interface to the system from the host PC.

Overall the system is successful as a bathymetry coprocessor operating at a data rate of approximately 250 kbps, which allows approximately one eighth of the sonar swath to be processed in real-time. To process the entire swath the interface to the PC will need to be upgraded, and the processing resources on the FPGA increased. However, this system provides a good base for the development of a fully functional system, and the required improvements are discussed in some detail at the end of this chapter.

6.1.1 System-on-chip

Other than developing a real-time bathymetry processor, the work carried out in this thesis provided a useful evaluation of SoC technology for digital signal processing applications. The SoC design approach aims to simplify the process of embedded system design by implementing the entire system inside the FPGA through the use of IP cores, which form the building blocks of the system. Within SoC, the designer has the choice

between two design routes – using open source cores or proprietary IP cores – which determine the reliability, design time, and cost of the system.

The two major programmable logic companies, Xilinx and Altera, both offer proprietary SoC systems featuring basic cores that are free to use once the SoC tools package has been purchased. More specialised cores can be purchased from third party vendors, but these are typically prohibitively expensive for academic work. An open source community exists with the aim of creating a library of reusable FPGA and ASIC cores, but unlike proprietary cores they are often unverified in hardware or only verified for one device. The level of documentation and support is also often somewhat less than that offered with a proprietary solution. These drawbacks mean that an open source SoC design will typically require more development time than a design using proprietary IP, but has the advantage of low cost and complete re-usability. In this project a design comprising both proprietary and open source elements was used with good results. This approach is relatively low-cost and uses proprietary IP and tools to develop the basic system infrastructure, then uses open source cores and custom logic to add extra functionality. The non-proprietary components of the system must be wrapped to be compatible with the bus architecture, but tools are provided to simplify this process.

The SoC design methodology offers many advantages over traditional embedded design, the greatest of these being the level of flexibility offered. As the bulk of the system is contained within the FPGA, major modifications can easily be made without dramatically increasing the design time. For example, adding a component to a traditional design involves modifying the PCB, while the same change in a SoC design merely requires that the design be re-synthesised. In a traditional design the processor is unlikely to be changed once it has been chosen; in a SoC design modules can be added and removed from the processor as they are required.

6.2 FUTURE WORK

As the system is a proof-of-concept at this stage, there is plenty of scope for it to be improved. These improvements can be broken down into short-term, medium-term and long-term tasks.

6.2.1 Short-term

There are two pressing improvements to be made to the system. The first is to solve the clock jitter problem so that the USB core operates reliably. The simplest way to do this is to source a 48 MHz, 2.5 V LVTTL oscillator to use in the development board's secondary clock socket. This will eliminate the need to use frequency synthesis to generate the clock, resulting in a lower level of jitter that is within the USB specification.

The second is to alter the way that the magnitude of correlation peak is found. While quadratic interpolation will give results accurate enough to be used in the next stage of delay estimation, the current method has insufficient dynamic range to calculate the result. This could be remedied by including the quadratic fitting process as part of the coprocessor, as a custom hardware design will allow for extra guard bits to preserve the intermediate results in the algorithm.

Once the USB is operating more reliably, modifications should be made to the system to increase the processing rate. This could be achieved by speeding up the correlation process by either reducing the number of lags in the correlation or running the correlator at a higher clock speed. These changes should speed up the correlation operation enough to maximise the available USB 1.1 bandwidth, but the MicroBlaze code will likely become the bottleneck in the system because of the extra demands placed on it by the increased USB throughput.

6.2.2 Medium-term

The main goal in the medium-term should be the realisation of a system that performs at a similar level to MATLAB routines in real-time. This will require some major modifications, most notably upgrading interface between the host PC and the FPGA to handle the higher data rate requirement. Some feasible protocols are discussed in Chapter 5, of these USB 2.0 offers the best balance between ease of implementation, ease of use and maximum data rate. USB 2.0 requires an external PHY to meet timing requirements, so a daughterboard will need to be designed to accommodate this — the development board's P160 expansion port would be an ideal interface for the daughterboard. The USB 1.1 controller core will require an upgrade to USB 2.0, and should be modified to add DMA functionality.

The MicroBlaze processor should be replaced with the PowerPC 405 core, allowing faster execution of code and releasing device area for the implementation of additional coprocessors. This will require that the correlator block be wrapped for the PLB rather than the FSL, and should also be optimised by adopting a parallel implementation and pipeline stages to reduce the execution time. The correlation should also be unbiased as described in section 5.1.2. To increase the accuracy of the time delay estimate, it is desirable to average the correlation results across adjacent pings before the peak detection stage. The most efficient implementation to achieve this is an area requiring additional research, one possible avenue would be to process a number of pings in parallel using multiple coprocessors and implement some scheme to allow the coprocessors to communicate directly with one another.

6.2.3 Long-term

In the long-term there is scope for the bathymetry processor to be integrated into the towfish system as a PCI expansion card. The processor would then stream processed data up the Ethernet link to the tow vessel for display. This would of course require the design of a high-speed PCB with a larger FPGA device capable of handling the processing required prior to the interferometry stage as well. A chip such as the XC2VP30 or V4FX60 with two PowerPC cores would probably be suitable. Another interesting possibility is the new Virtex-5 chip, on which a MicroBlaze processor can be implemented using only 2% of the device area, compared to 20% on the current device. A system comprising a large number of MicroBlaze cores operating in a distributed computing configuration via FSLs could be implemented, and is certainly worthy of investigation.

REFERENCES

[1] P. J. Barclay, *Interferometric Synthetic Aperture Sonar Design and Perfomance*. PhD thesis, University of Canterbury, 2006.

[2] A. Jongenelen, A. Coulson, and D. Carnegie, "Analysis of Fixed and Floating-Point Firmware Implementations of 802.11a Packet Detection," in *ENZCon'06 Proceedings*, 2006.

[3] Xilinx, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2005. `http://www.xilinx.com/bvdocs/publications/ds083.pdf`.

[4] "PicoBlaze 8-bit Microcontroller for CPLD Devices." Application Note, 2003. `http://www.xilinx.com/bvdocs/appnotes/`.

[5] Xilinx, *MicroBlaze Processor Reference Guide*, 2006. `http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf`.

[6] "Meet the PowerPC 405 Evaluation Kit: SoC Integration using the PowerPC 405 Architecture." Webpage, 2005. `http://www-128.ibm.com/developerworks/power/library/pa-pek/`.

[7] "OpenRISC 1200 Project." `http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200`.

[8] H.-P. Rosinger, "Connecting Customised IP to the MicroBlaze Soft Processor using the Fast Simplex Link (FSL) Channel." Application Note, 2004. `http://www.xilinx.com/bvdocs/appnotes/`.

[9] "CoreConnect^{TM}Bus Architecture," 1999. `http://www.ibm.com`.

[10] W. D. Peterson, "WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores." Internet Resource, 2002. `http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf`.

[11] S. Banks, *Studies in High Resolution Synthetic Aperture Sonar*. PhD thesis, University College London, 2002.

[12] H. J. Callow, *Signal Processing for Synthetic Aperture Sonar Image Enhancement.* PhD thesis, University of Canterbury, 2003.

[13] R. J. Urick, *Principles of Underwater Sound.* McGraw Hill, 1975.

[14] E. N. Pilbrow, *Synthetic Aperture Sonar Micronavigation Using An Active Acoustic Beacon.* PhD thesis, University Of Canterbury, 2007.

[15] D. R. Jackson, K. L. Williams, E. I. Thorsos, and S. G. Kargl, "High-Frequency Subcritical Acoustic Penetration into a Sandy Sediment," *IEEE Journal of Oceanic Engineering*, July 2002.

[16] A. W. Rihaczek, *Principles of High Resolution Radar.* McGraw Hill, 1969.

[17] A. J. Hunter, *Underwater Acoustic Modelling for Syntheic Aperture Sonar.* PhD thesis, University of Canterbury, 2006.

[18] J. Chatillon, A. E. Adams, M. A. Lawlor, and M. E. Zakharia, "SAMI: A Low-Frequency Prototype for Mapping and Imaging of the Seabed by Means of Synthetic Aperture," *IEEE Journal of Oceanic Engineering*, January 1999.

[19] S. Qureshi, *Embedded Image Processing on the TMS320C6000^{TM}DSP: Examples in Code Composer Studio^{TM}and MATLAB.* Springer, 2006. pp. 83–90.

[20] M. P. Hayes, "Results of a Multiple-Baseline Interferometric Synthetic Aperture Sonar in Shallow Water," in *Image and Vision Computing New Zealand*, 2006.

[21] C. H. Knapp and G. C. Carter, "The Generalized Correlation Method for Estimation of Time Delay," *IEEE Transactions on Acoustics, Speech and Signal Processing*, August 1976.

[22] S. A. Fortune, *Phase Error Estimation for Synthetic Aperture Imagery.* PhD thesis, University of Canterbury, 2005.

[23] M. Silventoinen and T. Rantalainen, "Mobile Station Locating in GSM," in *IEEE Wireless Communication System Symposium*, 1995.

[24] A. Gustavsson, P. O. Frolind, H. Hellsten, T. Jonsson, B. Larsson, and G. Stenstrom, "The Airborne VHF SAR System CARABAS," in *International Geoscience and Remote Sensing Symposium*, 1993.

[25] A. Kumar and Y. Bar-Shalom, "Time-Domain Analysis of Cross Correlation for Time Delay Estimation with an Autocorrelated Signal," *IEEE Transactions on Signal Processing*, April 1993.

[26] L. Sha and B. L. F. Daku, "A Performance Comparison of Three Localization Algorithms," in *Canadian Conference on Electrical and Computer Engineering*, 2005.

[27] A. M. Rincon, W. R. Lee, and M. Slattery, "The Changing Landscape of System-on-a-Chip Design," in *Proc. IEEE 1999 Custom Integrated Circuits Conf*, pp. 83–90, 1999.

[28] K. Zhu and D. H. Wong, "Clock Skew Minimisation During FPGA Placement," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, April 1997.

[29] K. N. Macpherson and R. W. Stewart, "Low FPGA Area Multiplier Blocks for Full Parallel FIR Filters," in *Proceedings of IEEE International Conference on Field Programmable Technology*, 2004.

[30] D. Kim, "An Implementation of Fuzzy Logic Controller on the Reconfigurable FPGA System," *IEEE Transactions on Industrial Electronics*, June 2000.

[31] N. S. Stolton and J. D. Provence, "Measures of Syntactic Complexity for Modelling Behavioural VHDL," in *Proceedings of 32nd IEEE Conference on Design Automation*, 1995.

[32] P. R. Panda, "SystemC — A Modelling Platform Supporting Multiple Design Abstractions," in *The 14th International Symposium on System Synthesis*, 2001.

[33] K. Parnell and R. Bryner, "Comparing and Contrasting FPGA and Microprocessor System Design and Development," Tech. Rep. White Paper 213, Xilinx, 2004. `http://direct.xilinx.com/bvdocs/whitepapers/wp213.pdf`.

[34] "OpenCores Open Source IP Repository." `http://www.opencores.org`.

[35] Xilinx, *PowerPC 405 Processor Block Reference Guide*, 2005. `http://www.xilinx.com/bvdocs/userguides/ug018.pdf`.

[36] "PicoBlaze Integrated Development Environment." `http://www.mediatronix.com/pBlazeIDE.htm`.

[37] *Universal Serial Bus Specification*, 2000. `http://www.usb.org/developers/docs`.

[38] "USB in a Nutshell." `http://www.beyondlogic.org/usbnutshell/`.

[39] P. Alfke, "Asynchronous FIFO in Virtex-II^{TM} FPGAs." Internet Resource, 2001. `http://www.xilinx.com`.

[40] Xilinx, *Designing Custom OPB Slave Peripherals for MicroBlaze*, 2002. `http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/opb_tutorial.pdf`.

[41] P. T. Gough, "Signal processing and correlation techniques." http://pollux.dhcp.uia.mx/manuales/Filtros/UIA_correlation.pdf.

[42] K. Turkowski, "Fixed Point Square Root," tech. rep., Advanced Technology Group, Apple Computer Inc., 1994. http://www.worldserver.com/turk/computergraphics/FixedSqrt.pdf.

[43] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design.* Wiley, 1979. pp. 360–379.

[44] D. G. Bailey, "Space Efficient Division on FPGAs," in *ENZCon'06 Proceedings*, 2006.

[45] "Virtex-II CLKFX Jitter Calculator." Webpage, 2007. http://www.xilinx.com/applications/web_ds_v2/jitter_calc.htm.